Design and analysis of cryptographic algorithms

Kölbl, Stefan

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

STEFAN KÖLBL

DESIGN AND ANALYSIS OF CRYPTOGRAPHIC ALGORITHMS

DESIGN AND ANALYSIS OF CRYPTOGRAPHIC ALGORITHMS

STEFAN KÖLBL

Ph.D Thesis

September 2016

Supervisor: Christian Rechberger
Co-supervisor: Lars R. Knudsen

Technical University of Denmark
DTU Compute, Cyber Security

Abstract

In today's world computers are ubiquitous. They can be found in virtually any industry and most households own at least one personal computer or have a mobile phone. Apart from these fairly large and complex devices, we also see computers on a much smaller scale appear in everyday objects in the form of micro-controllers and RFID chips.

What truly transformed our society are large scale networks, like the Internet or mobile telephone networks, which can link billions of devices. Our ways of communicating and conducting business have severely changed over the last decades due to this development. However, most of this communication happens over inherently insecure channels requiring methods to protect our communication. A further issue is the vast amount of data generated, which raises serious privacy concerns.

Cryptography provides the key components for protecting our communication. From securing our passwords and personal data to protecting mobile communication from eavesdroppers and our electronic bank transactions from manipulation. These applications would be impossible without cryptography.

The main topic of this thesis is the design and security analysis of the most fundamental algorithms used in cryptography, namely block ciphers and cryptographic hash functions. These algorithms are the building blocks for a vast amount of applications and play a vital role in providing both confidentiality and integrity for our communication.

This work is organized in two parts. First, an introduction to block ciphers and cryptographic hash functions is given to provide an overview over the state-of-the-art, the terminology, and how we can evaluate the security of an algorithm. The second part is a collection of scientific publications that have been written during the PhD studies and published.

In the first publication we analyze the security of cryptographic hash functions based on the AES and demonstrate practical attacks on reduced-round versions of these algorithms. The second publication provides cryptanalysis of the lightweight block cipher SIMON in particular how resistant this type of block ciphers are against differential and linear cryptanalyis. In the fourth publication we present a short-input hash function utilizing AES-specific instructions on modern CPUs in order to improve the performance of hash-based signature schemes. The last publication deals with the design of the tweakable lightweight block cipher Skinny which provides strong security bounds against differential and linear attacks while also competing with the performance of SIMON.

Resumé

I nutidens verden er computere allestedsnærværende. De findes inden for enhver industri, og de fleste husholdninger har mindst en personlig computer eller en mobiltelefon. Ud over disse forholdsvist store og komplekse enheder, begynder computere i en meget mindre skala også at dukke op i hverdagen i form af mikrocontrollere og RFID-chips.

Det der for alvor har ændret vores samfund er netværk i stor skala, såsom Internettet eller mobilnetværker, som kan forbinde milliarder af enheder. Den måde hvorpå vi kommunikerer og gør forretning har ændret sig voldsomt de sidste par årtier, netop på grund af denne udvikling. Størstedelen af denne kommunikation foregår imidlertid over usikre kommunikationskanaler, hvor metoder til beskyttelse af kommunikationen er påkrævet. Derudover bliver store mænger data genereret, hvilket giver anledning til bekymringer om vores privatliv.

Kryptologi leverer det centrale element i beskyttelsen af vores kommunikation. Fra sikring af passwords og personlig data, til forebyggelse af aflytning af mobilkommunikation og manipulation af bankoverførelser - uden kryptologi ville alt dette være umuligt.

Hovedemnet i denne afhandling er design og sikkerhedsanalyse af de mest fundamentale algoritmer, der bliver benyttet i kryptologi: block ciphers og kryptografiske hash funktioner. Disse algoritmer er byggesten i mange anvendelser og spiller en afgørende roller i at levere både fortrolig kommunikation og dataintegritet.

Afhandlingen består af to dele. Første del giver en introduktion til block ciphers og kryptografiske hash funktioner med det formål at give et overblik over state-of-the-art, terminologien, og hvordan vi kan evaluere en algoritmes sikkerhed. Den anden del er en samling af videnskabelige publikationer, der er blevet skrevet og udgivet under PhD-studiet.

I den første publikation analyserer vi sikkerheden af kryptografiske hash funktioner baseret på AES og demonstrerer praktiske angreb på versioner af disse algoritmer med et reduceret antal runder. Den anden publikation indeholder kryptoanalyse af letvægts block cipheren SIMON med fokus på hvor resistent denne type block cipher er over for differentiel og lineær kryptoanalyse. I den fjerne publikation præsenterer vi en hash funktion med kort input, der benytter AES-specifikke instruktioner på moderne CPU'er for at forbedre hash-baserede signatureres ydeevne. Den sidste publikation omhandler designet af en tweakable letvægts block cipher, Skinny, som giver stærke sikkerhedsgarantier mod differentiale og lineære angreb, men hvis ydeevne stadig er sammenlignelig med SIMONs ydeevne.

Acknowledgments

First of all, I would like to thank my supervisor Christian Rechberger for his guidance throughout my PhD studies, pointing out interesting research problems and making this three years of PhD a very enjoyable experience. I would also like to thank Lars R. Knudsen, my co-supervisor and head of the research group, who provided an excellent working environment and always had an open door.

Thanks to all my co-workers at the DTU Cyber Security group: Mohamed Ahmed Abdelraheem, Hoda A. Alkhzaimi, Subhadeep Banik, Andrey Bogdanov, Christian D. Jensen, Martin M. Lauridsen, Arnab Roy, Elmar Tischhauser, Philip Vejre and our secretary Ann-Cathrin Dunker. Especially, to my PhD colleagues Martin, Philip and Tyge for all the interesting discussions on cryptography, life and other nonsense.

I would also like to thank all the great people in the crypto community whom I met at conferences and helped broadening my knowledge. In particular my co-authors: Ralph Ankele, Christof Beierle, Jérémy Jean, Martin M. Lauridsen, Gregor Leander, Florian Mendel, Amir Moradi, Tomislav Nad, Thomas Peyrin, Christian Rechberger, Arnab Roy, Yu Sasaki, Pascal Sasdrich, Martin Schläffer, Siang Meng Sim and Tyge Tiessen.

I would also like to thank the members of the research group at NTU, especially my host Thomas Peyrin. I truly enjoyed working with all of you and had a great time in Singapore.

Finally, I would like to thank all my friends in Austria and my family for all the support and encouragment.

Contents

Part I

SYMMETRIC PRIMITIVES

This part gives a concise introduction to both block ciphers and cryptographic hash functions. This includes different notions of security, applications and also techniques to study the security of these algorithms.

1

Introduction

Cryptography studies techniques for secure communication over an insecure channel and has a long and interesting history. In some forms it has already been used in ancient times, like the famous ancient *Caesar cipher* (after Julius Caesar), which just consisted of shifting the position of the letters in the alphabet to hide the original message. Another famous historical example is the break of the German Enigma machine by Polish and British cryptologists. This allowed the western Allies to read a large amount of messages transmitted during World War II, providing them a significant advantage. A comprehensive guide to the fascinating early history of cryptography is given by David Kahn in [37].

Before the information age cryptography mostly dealt with keeping messages secret and was only used by very few people like diplomats, spies or the military. Only in the 1970's cryptography started attracting open academic research with the publication of the Data Encryption Standard (DES) and the invention of *public key* cryptography.

Computers and large scale networks have become ubiquitous today, with billions of users communicating and having billions of users and cryptography plays essential in keeping us safe. The nature of our modern communication channels is often inherently insecure and it is very easy for an adversary to eavesdrop a conversation (see Figure 1). Transmissions over radio waves, like they are used in our mobile communication or wireless networks, can easily be captured and are prone to manipulation. The infrastructure of the Internet is controlled by a vast amount of entities which are spread globally, offering an adversary a huge attack surface and we require ways to protect our private data, personal communication, electronic commerce and critical infrastructure.

While most people associate cryptography with keeping information secret, it serves various purposes nowadays. In practice it is not sufficient to just keep the messages transmitted secret, but we also want to make sure that they are not modified by a third party or that someone impersonates our communication partner. Through the research in cryptography many new applications emerged ranging from digital signatures, payment schemes like Bitcoin to secure multi-party computation (MPC) [77] and zero-knowledge proofs [30].

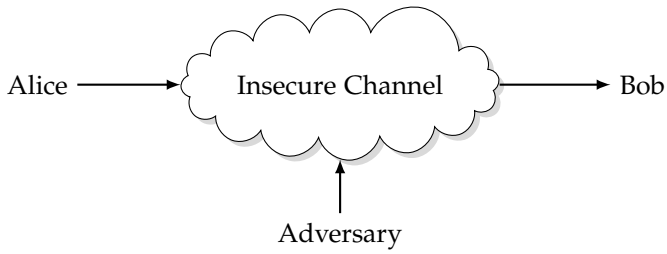


Figure 1: Two parties communicating over an insecure channel like the Internet or mobile networks. An adversary can listen to any communication on the channel, delete messages or modify them.

This work focus on how we can create a secure channel between two parties. We define the following three requirements for this:

- **Confidentiality:** When an adversary listens to the communication she should not be able to derive any information on the messages being exchanged between the two parties.
- **Integrity:** When an adversary modifies the transmitted messages the communicating parties should be able to detect that a modification has occurred.
- **Authenticity:** The communicating parties should be able to verify that the message originated from whom they expect. It should not be possible for the adversary to impersonate as one of the persons communicating.

In this thesis we will look at different cryptographic algorithms which provide one or more of these properties. The first class of algorithms are *block ciphers* which use a *secret key*, which is shared between the two parties, to encrypt a message which can be send over an insecure channel without loss of confidentiality. The second class of algorithms are *cryptographic hash functions*, which provide integrity by giving the two parties a way to detect any potential malicious changes to messages transmitted. We can also use these functions in combination with a secret key to authenticate messages, as will be shown later.

What is not covered in this thesis are so called *public key* algorithms. The main advantage of these algorithms is that they do not require any prior shared secret between the two parties. They involve a pair of keys, the private key and the public key, where the latter be made public. Using this public key anybody can encrypt a message, which can only be decrypted if one knows the corresponding private key. While the private key and the

public key are related, it should be infeasible for an adversary to recover it. These algorithms are often based on *hard problems* like *integer factorization* (RSA) or *discrete logarithms*. A major drawback of these algorithms is that they are much slower then symmetric key algorithms and therefore in practice, protocols or applications usually use a combination of these algorithms. The public key algorithms are used to exchange keys, while the large bulk of data is then encrypted with the much faster symmetric algorithms.

Outline. In Chapter 2 we give a short introduction to block ciphers. This includes how we can argue about security for these algorithms, the capabilities of an adversary and some basic design principles on how they are constructed in practice. In Chapter 3 we discuss the same aspects for cryptographic hash functions and also give a more detailed discussion of some applications with a focus on hash-based signature schemes. At last, in Chapter 4 we discuss some of the most prevalent cryptanalysis techniques for both block ciphers and hash functions, namely meet-in-the-middle attacks and differential cryptanalysis.

2

Block Ciphers

A *block cipher* is a family of functions parameterized by a *key* $K \in \{0,1\}^k$, which maps a set of *messages* $M \in \{0,1\}^n$ to a set of *ciphertexts* $C \in \{0,1\}^n$

$$\mathcal{E} : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n \tag{1}$$

We will use the key as a subscript $\mathcal{E}_K(M)$ for the encryption of M using the key K. The inverse of this operation $\mathcal{D}_K = \mathcal{E}_K^{-1}$ is called the decryption algorithm. $\mathcal{K}$ is the set of possible keys, $\mathcal{M}$ the set of possible messages and $\mathcal{C}$ the set of possible ciphertexts. We refer to k as the *key size* and n as the *block size*.

The main purpose of a block cipher is to provide confidentiality. We can transmit the ciphertext over any insecure channel and it should be impossible for any third party to derive information of the plaintext without knowing the secret key.

The first standardized block ciphers is the *Data Encryption Standard* (DES), which was published in 1975 and standardized in 1977. DES evolved out of an earlier design by Horst Feistel developed at IBM and modified in consultation with the *National Security Agency* (NSA). It was quickly adopted internationally and attracted lot of academic research. The study of DES has led to many advances in the cryptanalysis of block ciphers, which still influence designs nowadays.

As DES was aging and only had a limited key size, the National Institute of Standards and Technology decided to hold a new and more open competition to find a successor, the Advanced Encryption Standard (AES). The competition was lasting from 1997 to 2000 and received a lot of attention from the cryptographic research community. Fifteen block cipher designs were submitted both by companies and researchers. In October 2000 the winner *Rijndael* by the Belgian cryptographers Joan Daemen and Vincent Rijmen was announced [72]. Since then the algorithm became widely spread and modern CPUs even provide instructions specific to the AES.

There are several areas emerging in which highly resource constraint devices are used to build networks and communicate with each other, for example RFID chips or sensor networks. For these platforms algorithms like the AES are not always suitable due to the limited chip-area, code-size or strict requirements on the latency. This has lately been reflected by various

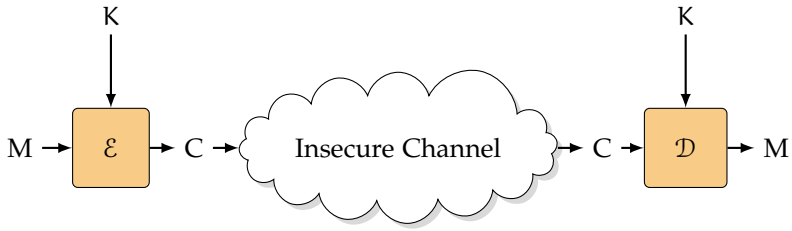


Figure 2: A block cipher can use a shared secret K to establish confidentiality over an insecure channel. The encrypted messages $C = \mathcal{E}_k(M)$ can be send over the insecure channel and gives an adversary no information on M.

lightweight designs, which optimize for several of these criteria and address this restrictions. Examples include Noekeon [18], Present [13], Prince [14], SIMON [4], Speck [4] and also the lightweight cipher Skinny [5] which will later be discussed in this thesis.

2.1 Applications

Block ciphers are used in various protocols to provide confidentiality and are used to encrypt the main bulk of data. This includes encrypting traffic over the Internet, which is usually done using TLS [23], single files or providing full disk encryption for your computer. They are also an elemental building block for other cryptographic primitives including *stream ciphers*, *message authentication codes* (MACs), *pseudo-random number generators* and *hash functions*.

2.2 Security

The main purpose of a block cipher is to provide *secure* communication between two parties. Giving an exact and concise definition of *secure* is a difficult task and depends both on the requirements of the parties communicating and the goals and capabilities of the adversary.

2.2.1 Unconditional Security

We call an encryption system unconditionally secure if it can not be broken by an adversary, even if she has unlimited computational resources available [24]. This property results from having multiple meaningful solutions for a given ciphertext, which are all equally likely to be the secret message. For example, if the ciphertext `JxkMc` could be decrypted to `mouse`, `sloth`,

`tiger`, ... for different keys then the adversary can not distinguish anymore which is the right solution. Shannon [67] used the term *perfect secrecy* for a similar definition

Definition 2.1. A system has *perfect secrecy* if for any message $m \in \mathcal{M}$ and ciphertext $c \in \mathcal{C}$

$$\Pr(m \mid c) = \Pr(m). \tag{2}$$

Note that this implies that for any pair of (m, c) there must be at least one key $k \in \mathcal{K}$ which relates these two values $c = \mathcal{E}_K(m)$. Therefore the key space $\mathcal{K}$ must be at least as big as the message space.

Can we now realize such a system in practice? In fact, there is a very simple encryption scheme which achieves this property the *one-time pad*.

One-time Pad For the one-time pad we have $\mathcal{M} = \mathcal{C} = \mathcal{K} \in \mathbb{F}_2^n$. To encrypt a message $m = (m_1, \ldots, m_n) \in \mathcal{M}$ we select a key $k = (k_1, \ldots, k_n) \in \mathcal{K}$ uniformly at random. The encryption function is then given by

$$\mathcal{E}_K(m) = m \oplus k = (m_1 \oplus k_1, m_2 \oplus k_2, \ldots, m_n \oplus k_n) \tag{3}$$

While this scheme achieves perfect secrecy it has mayor limitations. First, the key has to be as long as the message and second it can only be used once without compromising the security. Encrypting a second message m' with the same key would allow an attacker to learn $m \oplus k \oplus m' \oplus k = m \oplus m'$ and therefore leak information on the plaintext.

2.2.2 Computational Security

While the previous schemes are provably secure they have severe drawbacks in practice due to the size of the key or requiring a new key for each encryption. Furthermore, in practice an attacker is always limited by the available amount of computing power, memory storage and amount of messages she is able to obtain. Hence, for encryption schemes used in practice it is reasonable to look at *computational* secure systems.

The size of the key used in a block cipher is always limited and therefore an attacker can always succeed in finding the secret key by searching through the whole key space. This *exhaustive search* gives an inherent upper limit to the security one can expect from a block cipher and also requires the keys to be sufficiently large, typically 128 or 256 bits in practice.

We say a block cipher provides t-bit security if the best attack requires an exhaustive search over 2^t values. In general it is very difficult to prove that a cipher is computational secure and the usual approach is to show that it provides sufficient security against all known attacks, that means no attack is faster then exhaustive search.

2.2.3 Adversary Goals

From an attacker's point of view recovering the secret key is the best case, however in practice it is not sufficient to only protect against this type of attack. We also have to consider scenarios which appear less severe at first sight and consider other goals which an adversary might want to achieve. In the following we give a list of those in decreasing order of severity [42].

Total Break The attacker fully recovers the secret key K.

Global Deduction The attacker finds an algorithm which is functionally equivalent to $\mathcal{E}_K$ or $\mathcal{D}_K$. This would allow her to decrypt any messages, but would not compromise the choice of K.

Local Deduction The attacker can generate M (or C) corresponding to a C (or M) which has not been seen before. In this case the attacker might only be able to derive information of a single intercepted ciphertext.

Distinguisher The attacker is given access to either the block cipher using a random key or a randomly chosen mapping from $\mathcal{M}$ to $\mathcal{C}$. The goal for her is to decide whether a given output C is coming from the block cipher or the random mapping.

2.2.4 Capabilities of the Attacker

In order to break a cryptosystem the attacker first has to assess which algorithms are used. Therefore, one might assume that keeping the algorithm secret will result in a more secure system, however this is rarely the case and can often be a dangerous pitfall giving a false sense of security. For all our cryptosystems we assume that the attacker has full knowledge of the underlying algorithms. In cryptography this is often referred to as *Kerckhoffs' principle* which states that the security of a cryptosystem should solely depend on the secrecy of the key.

Designing a cryptosystem under this assumption can be very beneficial and lead to a more robust system in general. It is easier to protect a small piece of information, the key, than the whole description of the algorithm. In practice an attacker will often have access to a system which implements the block cipher allowing to reverse engineer the process. Additionally, in the case of compromise it is much simpler to replace a key then replacing the algorithm in all implementations deployed, especially in Hardware. Another

important aspect is that we have a higher confidence in published cryptographic algorithms, as they have convincingly survived years of attack efforts by a large research community.

Traditionally, when you think about breaking encryption, you imagine intercepting some *scrambled* message and trying to deduct the secret message which has been transmitted. However, this is not the only relevant scenario in practice and an attacker might be able not only to passively listen on an encrypted channel, but also send her own messages or modify messages being send. In the following we give an overview over the most relevant ways an attacker can obtain data, starting from the one were she has least control over:

- **Ciphertext only**: The attacker intercepts a set of messages $C_1, \ldots, C_n$ encrypted with the key K. Note that the attacker might still infer some knowledge on the plaintexts, for instance the language used.
- **Known-plaintext**: The attacker is able to obtain a set of ciphertexts $C_1, \ldots, C_n$ encrypted with K where she knows the corresponding messages $M_1, \ldots, M_n$. A typical example for this would be fixed headers for instance of an e-mail or TCP/IP package.
- **Chosen-plaintext**: The attacker can request the encryption with key K for a set of messages $M_1, \ldots, M_n$ of her choice. While at first this might seem like an artificial scenario it can be useful in practice. A historical example for this can be found during World War II, where the British RAF would drop mines in specific locations in order to induce warning messages with a specific content being transmitted.
- **Adaptive chosen-plaintext**: In this scenario an attacker can obtain the encryption of messages in an interactive way. This means she can request $C_i = \mathcal{E}_K(M_i)$ and depending on the outcome of this issue a new request for C_{i+1}.

When discussing the actual costs of an algorithm breaking an encryption scheme we will use the following three metrics:

- **Time T**: This represents the computational effort required to break the system. For practical attacks this could be stated as the actual time it requires to recover the key. However as this strongly depends on the computational power available it is more common to express T as the number of *simple* operations required to succeed. This can be single instructions on a CPU or the number of calls to the block cipher.

To put this into context, a typical processor found in a consumer PC can do $\approx 2^{32}$ operations per second. The Bitcoin network manages to compute $\approx 2^{60}$ hash computations per second[1].

- **Memory M**: The storage space required for an attack is equally important and can severely hinder the practicality of an attack. Accessing the memory usually also gets more costly with the amount of memory required. While a typical hard drive nowadays offers 2^{42} bits of memory, accessing it is significantly slower than the main memory or cache of a CPU.
- **Data D**: The amount of data required for an successful attack. In practice it might be very difficult to obtain a large amount of data, especially if an attack requires very specific messages.

Independent of the structure of a block cipher an attacker can always apply *generic attacks*. As no block cipher can be secure against these attacks they provide an upper bound on what level of security one can expect. These attacks typically only depend on the size of the key or the block size. In general we consider a cipher *broken* if there exists an attack which is more efficient than the best known generic attack. In order to prevent these generic attacks in practice the key size should not be below 128 bits resp. 256 bits in the quantum computing settings. Here, we briefly state two of the most important generic attacks.

Exhaustive Search The attacker tries out all the possible keys. For a key-size of k bits the time complexity is $\approx 2^k$. The attack requires no memory and only requires a few plaintext/ciphertext pairs depending on the block size and unicity distance of the underlying messages [50]. In the quantum computing setting Grover's algorithm [31] is able to find a key in $2^{k/2}$ evaluations.

Time-memory Trade-off These type of attacks, which as the name suggests, allow a trade-off between the time and memory complexity by precomputing a table which then enables a faster key recovery attack. The variant proposed by Hellman [32] requires 2^k pre-computation and then can find the secret key with a total complexity $T \times M = 2^k$, where T is the time complexity and M the amount of memory used. While this approach might not be useful for finding a single key it provides a significant speed-up when attacking multiple targets.

[1] https://blockchain.info/charts/hash-rate on 22th July, 2016.

2.3 Design

A block cipher can be seen as a set of 2^k permutations on n-bit words which are indexed by the key k. Ideally we want a block cipher to randomly draw 2^k permutations out of the possible $2^n!$ permutations on n-bit words. Unfortunately, in practice such a *random* block cipher would be very difficult to implement and require a huge description for any meaningful key and block sizes. Therefore, in practice we need a more structured approach which gives us an *efficient* block cipher and still makes it difficult for an adversary to find any relationship between plaintext, ciphertext and the secret key.

Nowadays, most block cipher found in practice use simple building blocks which are combined to get a complex function. One of the first constructions of this kind, the *product cipher*, was described by Shannon in [67].

Definition 2.2. An *iterative block cipher* is an algorithm which maps a plaintext of fixed size n into a fixed size ciphertext n' using a key K, by repeatedly applying a *round transformation* f^r to the plaintext

$$\mathcal{E}_K(\cdot) = f^r_{k_r}(\cdot) \circ \ldots \circ f^1_{k_1}(\cdot) \tag{4}$$

The *round keys* $k_1, k_2, \ldots, k_r$ are derived from K by a so called *key schedule*. The intermediate outputs of the function are called *intermediate states*. If the round functions are all equal we also refer to this as an *iterated* block cipher.

The main difficulty for a designer is now to choose a round function and method to factor in the key to get a complex relationship between the plaintext, ciphertext and key. The two most common ways to achieve this are so-called Feistel networks or Substitution Permutation Networks (SPNs). Another less widespread way to design block cipher is the construction by Lay-Massey, which is used in IDEA [43].

2.3.1 Feistel

The first construction we will look at are so called Feistel networks, named after German cryptographer Horst Feistel. The basic idea of Feistel ciphers is to split the message in two parts and apply the round transformation only to one half. The result is then added using XOR to the other half and the two outputs are swapped (see Figure 3). The seminal work by Luby and Rackoff [47] showed that if the round transformation is a secure *pseudorandom function* then three rounds are sufficient to get a *pseudorandom permutation*.

There are several benefits by using this construction. First of all, the round transformation does not have to be invertible, giving the designer more options to choose from. Also the computationally expensive round transformation are only applied to half the state. However, this will usually require a

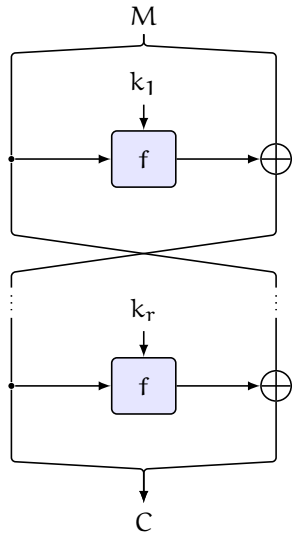


Figure 3: Outline of an r-round Feistel network.

higher number of rounds to account for. Furthermore, the decryption algorithm can easily be derived by changing the order of the round keys.

Apart from the *classic* construction, Feistel networks come in large variety of flavors. There are *unbalanced* Feistel networks [65], where the message is split in uneven parts, or Feistel networks with multiple branches as proposed in [78].

The most prominent Feistel cipher is the *Data Encryption Standard* (DES), first published in 1975 and has been a federal standard (FIPS-46) in the U.S. from 1977 until 2004 [69]. DES operates on 64-bit blocks and uses a 56-bit key, which has been controversial already since its initial publication, and can not be considered a secure block cipher anymore as it is vulnerable to brute-force attacks. However, a variant of DES, the *Triple*-DES which has a larger key size and provides a security level of around 112 bits is still in use and remains a standard (FIPS-46-3).

Feistel ciphers have also been a popular choice in the *Advanced Encryption Standard* competition, which was held by the *National Institute of Standards and Technology* (NIST) from 1997 to 2000 with the goal of finding a successor for DES. Out of the five finalists, three designs where based on Feistel networks (MARS [16], RC6 [60], Twofish [66]). Another example can be found in mobile communication systems like GSM and UMTS, which use the MISTY1/Kasumi [48] block cipher.

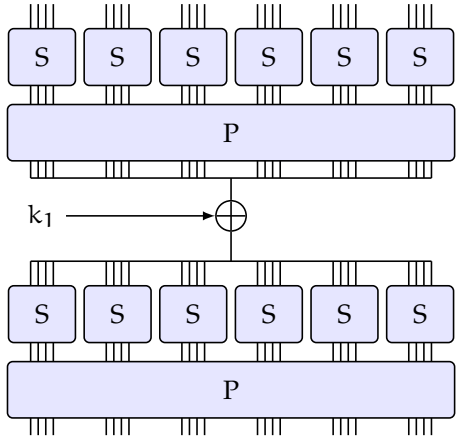


Figure 4: Outline of a substitution permutation network (SPN).

2.3.2 Substitution Permutation Networks

The second most important construction are *Substitution Permutation Networks* (SPNs). The round transformation is composed of a *substitution layer*, which applies a complex operation on small chunks of the state followed by applying a simpler *permutation layer* on larger parts of the state (see Figure 4).

The substitution layer applies multiple *substitution boxes* (S-boxes) to the state. These S-boxes provide a complex non-linear mixing on a small part of the state, typically 4 or 8 bits and can be efficiently realized with lookup tables. Designing S-boxes which help strengthen the security of the block cipher and are also efficient to implement in Hardware is an important research area.

The permutation layer operates on the whole state and the main purpose is to quickly diffuse the relationship between parts of the state. This is often realized by matrix multiplication or as a bit permutation, like in Present [13], which only requires rewiring in hardware.

The most important block cipher using this construction is *Rijndael*, now known as the *Advanced Encryption Standard* (AES) standardized by NIST in 2001 [72]. It uses 8-bit S-boxes, operates on blocks of 128-bit and supports key sizes of 128-, 192- and 256-bit. A detailed description of the AES and its design rationales can be found in [19].

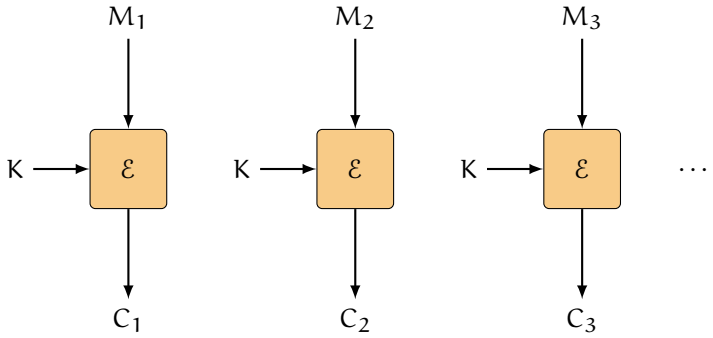


Figure 5: The electronic code book (ECB) mode encrypts each message block $M_1, \ldots, M_n$ individually.

2.4 Modes of Operation

A block cipher on its own can only encrypt rather short messages, with typical block sizes of 64 or 128 bits and therefore has limited use in practice. In order to encrypt arbitrary sized messages, various *modes of operation* exists with different properties regarding security, error propagation and implementation. In the following we will look at different ways to encrypt a message $M = M_1, \ldots, M_n$ and the advantages and disadvantages of the most commonly used modes.

2.4.1 Electronic Code Book

The *Electronic Code Book* (ECB) is the simplest mode where each block is encrypted independently of the others

$$C_i = \mathcal{E}_K(M_i) \quad \text{for} \quad 1 \leqslant i \leqslant n. \tag{5}$$

Both encryption and decryption are parallelizable and it also allows *random access* as one can decrypt any block independently from the others (see Figure 5). One of the main problems with ECB is that it can leak information, as the same message block encrypted with the same key will always result into identical ciphertext blocks. Therefore, it is not recommended to use this mode in practice for encrypting longer messages.

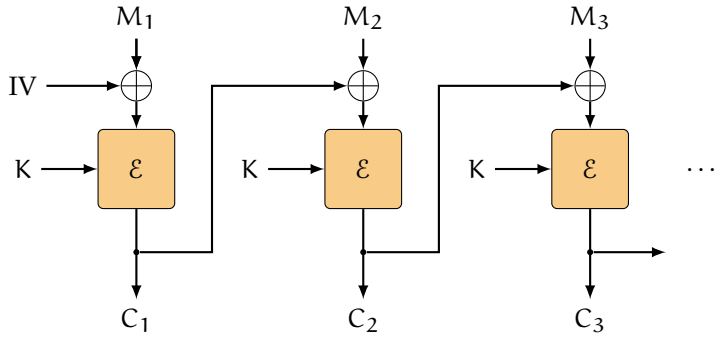


Figure 6: In the cipher block chaining (CBC) mode, each subsequently encrypted block depends on the previous blocks.

2.4.2 Cipher Block Chaining

The *Cipher Block Chaining* mode adds some dependency on the encryption of all previous blocks. The ciphertexts are computed as

$$C_1 = \mathcal{E}_K(M_1 \oplus IV) \quad \text{and} \quad C_i = \mathcal{E}_K(M_i \oplus C_{i-1}) \quad \text{for} \quad 2 \leqslant i \leqslant n. \tag{6}$$

The *initial value* (IV) has to be transmitted with the ciphertext to be able to decrypt the whole message and should be unpredictable. As the encryption of a block now depends on all previous blocks of ciphertext it is not possible to parallelize the encryption process.

While there is no obvious information leaking compared to ECB, it can still occur. Assume you have found two ciphertexts C_i, C_j for $i \neq j$ which are equal

$$C_i = C_j \to \mathcal{E}_K(M_i \oplus C_{i-1}) = \mathcal{E}_K(M_j \oplus C_{j-1}).$$

This can only hold if $M_i \oplus C_{i-1} = M_j \oplus C_{j-1}$ and we can therefore learn $M_i \oplus M_j$. For a block cipher with b-bit blocksize we expect this to occur after $\approx \sqrt{2^b}$ blocks due to the birthday paradox (see Section 3.2.1.2). While this problem is less severe as the one in the ECB mode, it can still be an issue for smaller block sizes like 64 bits. A further issue with CBC in practice can be that it is vulnerable to padding oracle attacks [73].

2.4.3 Counter Mode

The *Counter* mode builds a *stream cipher* out of a block cipher creating a *key stream* through encryption of an incrementing counter value

$$C_i = \mathcal{E}_K(N \oplus i) \oplus M_i \quad \text{for} \quad 1 \leqslant i \leqslant n. \tag{7}$$

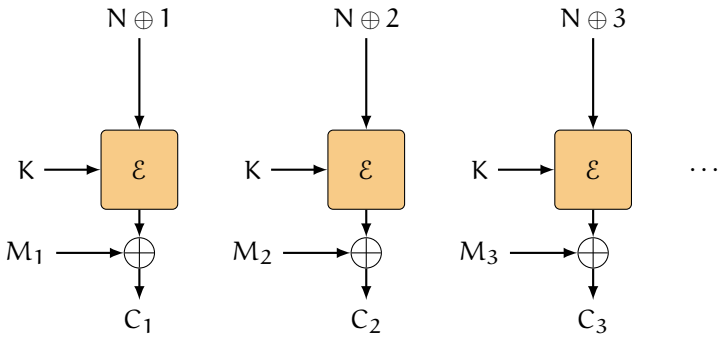


Figure 7: The counter (CTR) mode encrypts a unique value N and increasing counter to generate a continuous key stream.

There is a variety of options for the counter, typically it involves a *nonce* N, which is some unique number, and an integer which is incremented for each block. These two values can either be concatenated or XORed. It is important to never reuse the same key and nonce/counter combination, as otherwise this will result in the same key stream and allow an attacker to learn the XOR of the plaintexts similar to the issue we have seen with the one-time pad.

The CTR mode has various advantages over the previous modes. First, similar to ECB both encryption/decryption are parallelizable. CTR mode only requires the encryption function and can save the implementation costs of the decryption function of the block cipher. Additionally, the inputs are known in advance allowing to carry out preprocessing which can speed up the implementation.

A downside of the CTR mode is that the sender is stateful as she has to ensure that no counter is reused. However this is also often the case for CBC in practice when the IV is derived from the last encrypted block or chosen with some pseudorandom generator, to avoid the adversary predicting the IV.

2.5 Tweakable Block Ciphers

A *tweakable block cipher* [46] is a block cipher with the signature

$$\mathcal{E}_K : \{0,1\}^k \times \{0,1\}^t \times \{0,1\}^n \to \{0,1\}^n. \tag{8}$$

Here we call the second input the *tweak*, which is not secret and allows us to add some variability to the block cipher. A block cipher is deterministic and always gives the same output for a given message and key. However, this

property can be a problem in some modes of operations and applications which require a new instance of the block cipher for each call.

One of the main applications for a tweakable block ciphers is memory or disk encryption. The disk is usually organized in sectors of fixed size, typically 512 bytes and we want to be able to encrypt/decrypt those independently of all other sectors. A mode like CBC is unsuited for this application as it is not parallelizable and for CTR mode we require to never reuse the nonce/counter. A tweakable block cipher in ECB mode can provide a solution to this problem by using the sector index as a tweak, which guarantees that each sector is encrypted with a unique block cipher.

Modes like XE and XEX [62] can be used to turn any block cipher into a tweakable block cipher, but dedicated designs can be more efficient.

3

Hash Functions

A *cryptographic hash function* is an efficient deterministic algorithm, which maps messages of arbitrary length to strings of fixed length n

$$\mathcal{H} : \{0,1\}^* \to \{0,1\}^n. \tag{9}$$

The output of a hash function is called the *hash value*, *hash*, *digest* or *fingerprint* of a message. Every possible message is associated with such a value which can then be used as a short identifier or representative for this message. Cryptographic hash functions are used to provide integrity and authenticity in a large number of applications and protocols.

The first constructions of cryptographic hash functions were based on building a *compression function* from a block cipher. These functions have a fixed input size and can be used to build a hash function as shown independently by Merkle [52] and Damgård [21]. One of the first dedicated designs, MD4, was presented by Ron Rivest in 1990 [58], but quickly superseded by MD5 [59] due to security concerns. In 1993 NIST published SHA-0 [70] which has been designed by the NSA and follows a similar design strategy to MD5. It was withdrawn shortly after and SHA-1 was published [71] to correct a flaw in the design. Both MD5 and SHA-1 have been widely used for many years and still can be found in use today even though they are not considered secure anymore.

In 2004, weaknesses in several popular hash functions have been found leading to attacks on MD4, MD5 and SHA-1 [74–76]. Today, MD5 can be broken in seconds on a consumer machine and the attacks on SHA-1 are feasible for larger organizations, although no collision for SHA-1 has been published yet.

These attacks also casted doubt on the security of SHA-2, which follows a similar design strategy, and as a response NIST announced in 2007 that they will host a public competition, similar to the AES competition, to find a new cryptographic hash function standard, SHA-3. Fifty-one candidates were initially submitted and on October 2, 2012, Keccak designed by Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche was announced as the winner.

In the following, we will give a brief introduction to cryptographic hash functions, what they are used for and how we can construct them. In addition

we will have a look at what we mean by a *secure* hash function. For further reading on this topic a good starting point is [50].

We would also like to note that that there are non cryptographic hash functions, which are used for instance in data structures like hash tables. Although they provide similar functionality they do not require the same security properties we expect from cryptographic hash functions (see Section 3.2). In this work only the later are considered and therefore the prefix cryptographic will be usually omitted.

3.1 Applications

Cryptographic hash functions are one of the most versatile primitives and have become a fundamental part of modern cryptography. A common application for hash functions is to provide integrity for a message. If you flip a single bit in a message the resulting hash value will change allowing to detect any modifications.

One of the first applications of hash functions in cryptography was suggested by Rabin in 1978 [57]. As it is very expensive to digitally sign long messages he suggested to sign the hash instead. In signature schemes, like the digital signature algorithm (DSA), hash functions are used to get a short unique identifier for a message [27]. These schemes only sign the hash, which speeds up the computation while also providing additional security compared to using for instance raw RSA signatures. Hash functions can also directly be used to build signature schemes. We will discuss this application in more detail in Section 3.4.

A second important application are message authentication algorithms (MAC). A MAC is a keyed hash function that provides both integrity and authenticity of a message

$$\mathrm{MAC} : \{0,1\}^* \times \{0,1\}^k \to \{0,1\}^n. \tag{10}$$

We call the output *tag* and it can be used by the receiver to check if the message originates from the desired sender. An example for a widespread MAC is HMAC, used in standards like TLS [23] and IPSec [26].

Another common application is password protection. Passwords should not be stored as plaintext and a common practice is to only store the hash value. This allows the original password to be kept secret due to the one-wayness of hash functions. For a secure hash function it should be infeasible to derive a password from the stored hash-value. In general we want hash functions to be fast to compute, however in the case of password hashing the opposite is required. Password hashing and key derivation schemes like

bcrypt [55], scrypt [54] or Argon2 [11] are designed to be computationally expensive to make brute force attacks less efficient.

A further application is for confirmation of knowledge or commitment schemes. If you want to prove that you know some information, without revealing it, the hash of this information can be made public. Later you can prove that you had this information at the time of commitment by publishing the message and everyone can verify that it indeed corresponds to the previously published hash value.

3.2 Security

For a secure cryptographic hash function it should be difficult to find a message for a given hash value and it should also be difficult to find two messages which result in the same hash value, a *collision*. As the input domain of a hash function is always significant larger than the output domain this collisions are unavoidable.

We define the three main security requirements for a secure hash function as:

- **Preimage Resistance**: For a given output y it should be computationally infeasible to find an input x' such that $y = \mathcal{H}(x')$.
- **Second Preimage Resistance**: For a given x and $y = \mathcal{H}(x)$ it should be computationally infeasible to find $x' \neq x$ such that $\mathcal{H}(x') = y$.
- **Collision Resistance**: It should be computationally infeasible to find two distinct inputs x, x' such that $\mathcal{H}(x) = \mathcal{H}(x')$.

Note that collision resistance implies second preimage resistance. Assume $\mathcal{H}$ is collision resistant. If $\mathcal{H}$ is not second preimage resistant then it is possible to find, for a fixed x, a value $y = \mathcal{H}(x')$. Hence, (x, x') is a collision. However, collision resistance does not necessarily imply preimage resistance.

An ideal hash function should behave like a *random oracle*. A random oracle is a function which outputs a random value for each new input. If an input value is repeated it outputs the previously used value. No practical hash function can implement a random oracle, as the description would be too large. Nonetheless, a good hash function should be difficult for an attacker to distinguish from such a random oracle.

Similar to block ciphers we only require the hash function to be secure against a computational bound adversary. For hash function we only need to consider the time and memory complexity of an attack and the expected level of security is strongly correlated with the output size of the hash function n.

Table 1: The hash sizes for some of the most important hash functions over the last few years. Hash functions marked with † are insecure as there exist cryptanalytic attacks breaking the security claims.

Algorithm	Hash size	Year
MD4†	128-bit	1990
MD5†	128-bit	1992
SHA-1†	160-bit	1995
RIPEMD-160	160-bit	1996
Whirlpool	512-bit	2000
SHA-2	224-, 256-, 384- and 512-bit	2001
SHA-3	224-, 256-, 384- and 512-bit	2012

There exist also weaker variants of these properties which can be of interest. Near-collisions allow a *small* number of differences between the two hash values. Further examples include semi-free start and free-start collisions which allow an attacker to choose the initial value respectively choose two different initial values used for computing the hash of the colliding message pair. While such an attack has less practical impact it can still be useful to evaluate the quality of a hash function.

3.2.1 Generic Attacks

Similar to block ciphers, there exist also generic attacks on hash functions, which allow an attacker to find preimages or collisions disregarding the underlying structure. When treating the hash function as a black box the only relevant parameter for these attacks is the length of the hash value n. In Table 1 you can find the hash size of the most commonly used cryptographic hash functions and how the size evolved over the years.

3.2.1.1 Preimages

An attacker can always find a (second) preimage by trying out many inputs and checking whether they give the desired hash value. If the hash function has n-bit output then $\Pr(\mathcal{H}(X) = y) = 2^{-n}$, hence after trying $\approx 2^n$ inputs it is likely that an attacker succeeds in finding a preimage.

The only real improvement to this is if we give an attacker access to a quantum computer. In this case a preimage for a function f can be found in only $2^{n/2}$ steps using Grover's algorithm [31], which is also asymptotically optimal [6].

3.2.1.2 Collisions

Finding a collision for a hash function can be done more efficiently and for an output size of n bit a generic attack with a time complexity of $\approx 2^{n/2}$ exists. This fact is related to the *birthday paradox* which states that in a group of 23 people, the probability that two people share the same birthday is greater than 0.5.

When choosing j randomly distributed inputs to the hash function the probability that at least two inputs collide is

$$p_c = 1 - 1 \cdot (1 - \frac{1}{2^n}) \cdot (1 - \frac{2}{2^n}) \dots (1 - \frac{j-1}{2^n}) \approx 1 - e^{-\frac{j^2}{2^{n+1}}} \tag{11}$$

and hence we expect to find a collision after $\sqrt{\ln(2)2} \cdot 2^{n/2}$ trials.

The simplest attack using this property is to subsequently compute outputs for the hash function and store them in a list checking each time if the value is already in this list. While we can expect to find a collisions after the expected number of steps this method requires to store $\approx 2^{n/2}$ hash values, which would often be a bottleneck in practice. There exist memoryless versions of these *birthday attacks* based on cycle finding algorithms [56], which only have a very small memory footprint and are also parallelizable.

3.3 Design

Most hash functions follow an iterative design similar to Figure 8. The input m is split into evenly sized blocks $M_1, M_2, \dots M_n$ and a compression function f (see Section 3.3.2) is used iteratively to process each message block

$$h_1 = f(IV, M_1) \tag{12}$$

$$h_i = f(h_{i-1}, M_i) \qquad 1 \leqslant i \leqslant n \tag{13}$$

$$h = h_n \tag{14}$$

The initial value IV is some fixed constant and the intermediate values h_i are referred to as *chaining values*. The last chaining value h_n is used as the output of the hash function.

3.3.1 Merkle-Damgård construction

The Merkle-Damgård construction provides a method to construct a collision-resistant hash function from a collision-resistant compression function [21]. This construction is widely used and includes designs like MD5, SHA-1, SHA-2.

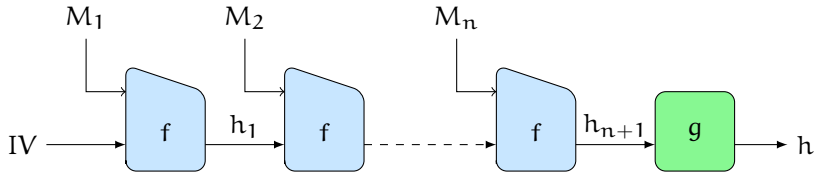


Figure 8: An iterative construction for a hash function. The message is split in evenly sized blocks $M_1, \ldots, M_n$ and processed using a compression function f. IV is a fixed constant and g the output transformation.

It uses an iterated construction based on a compression function f and adds the length of the message at the end of the padding which is referred to as Merkle-Damgård strengthening (MD-strengthening). It may also include an additional output transformation g.

The input M is padded by appending a 1-bit followed by the minimum number of 0 bits to result in a multiple of the block-size. This is followed by an additional block which encodes the binary representation of the length of M. This construction gives a provable collision-resistant hash function:

Proof. Let us assume that the hash function h is not collision-resistant and the attacker can find a colliding message pair (M, M') such that $h(M) = h(M')$. We consider the following two cases now:

- The length of the two messages is not equal. If the attacker has found a collision then the output of the last compression function call must be equal for a collision in the output. The last two message block contain the message length, therefore $M_n \neq M'_n$. However, this implies we have found a collision for the compression function $f(M_n, h_n) = f(M'_n, h'_n)$.
- The length of the two messages is equal. In this case at least one compression function call must lead to a collision $f(M_i, h_i) = f(M_j, h_j)$ for the output to be equal.

If an output transformation is applied then one has to consider collisions in the output transformation too. □

While this construction is provable collision-resistant it still has some properties which one would not expect from a secure cryptographic hash function.

An example for this is the length extension attack [17]. It allows an attacker to compute $\mathcal{H}(\text{pad}(M) \parallel X)$ without knowing M. This can be a problem, for instance if a MAC is constructed by computing $\mathcal{H}(key \parallel M)$. In this case it

would allow an attacker to forge valid tags for messages of the structure $\mathcal{H}(\text{pad}(key \parallel m) \parallel X)$.

Kelsey and Schneier showed that finding a second preimage for hash functions using the Merkle-Damgård construction is easier for large messages [38]. Using their results, a second preimage for SHA-1 for a message of size 2^{60} can be found with a complexity of 2^{106} compared to the costs of 2^{160} for the generic attack.

Computing multi-collisions, this means t messages $\mathcal{H}(M_1) = \mathcal{H}(M_2) = \ldots = \mathcal{H}(M_t)$, can be done more efficiently for hash functions based on the Merkle-Damgård construction. Joux presented an approach to construct 2^t-collisions at t times the costs of finding a collision for two messages [36].

3.3.2 Compression Functions

Compression functions

$$f : \{0,1\}^m \to \{0,1\}^n \qquad m > n. \tag{15}$$

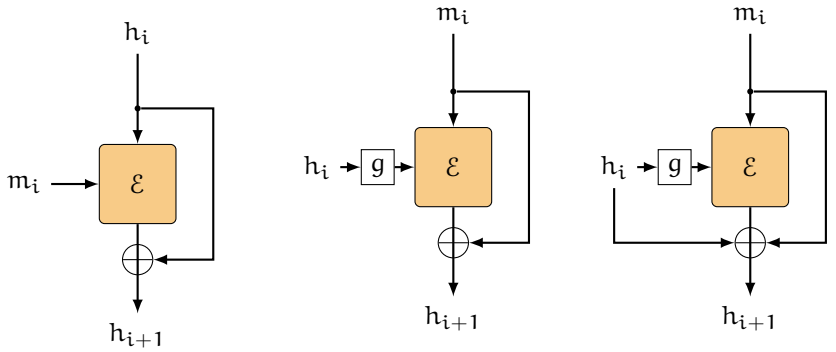
play an important role in the construction of cryptographic hash functions. The first designs were based on block ciphers, where for instance the message block is used as input for the key. There are various ways to construct a compression function from a block cipher (see Figure 9). One of the main drawbacks of this is that block ciphers typically only have a block size of 128 bits, leading to a 128-bit hash size which is not large enough for a secure hash function nowadays.

There are some constructions which address this problem, like the double length constructions MDC-2, MDC-4 and Hirose [33]. Nonetheless, the most popular constructions today are based on compression functions designed explicitly for hash functions which leads to more efficient solutions in practice.

The MD-family (MD4, MD5, SHA-1, SHA-2) share a similar design strategy for their compression function, with the main difference being an increased digest size, number of rounds and complexity of the round function. As an example how the round function is constructed we can look at SHA-1 (see Figure 10). The 160-bit state consists of five 32-bit words and in each round only one of them is updated. The 512-bit message block is expanded into 80 32-bit words W_i. This message expansion is very critical and has lead to security issues for the first published version (SHA-0).

3.3.3 Sponge construction

A fairly new design strategy based on permutations has become very popular recently and is also the basis for SHA-3. The *sponge construction* is a mode



(a) Davies-Meyer (DM)

(b) Matyas-Meyer-Oseas (MMO)

(c) Miyaguchi-Preneel (MP)

Figure 9: The three most commonly used ways to construct a compression function from a block cipher.

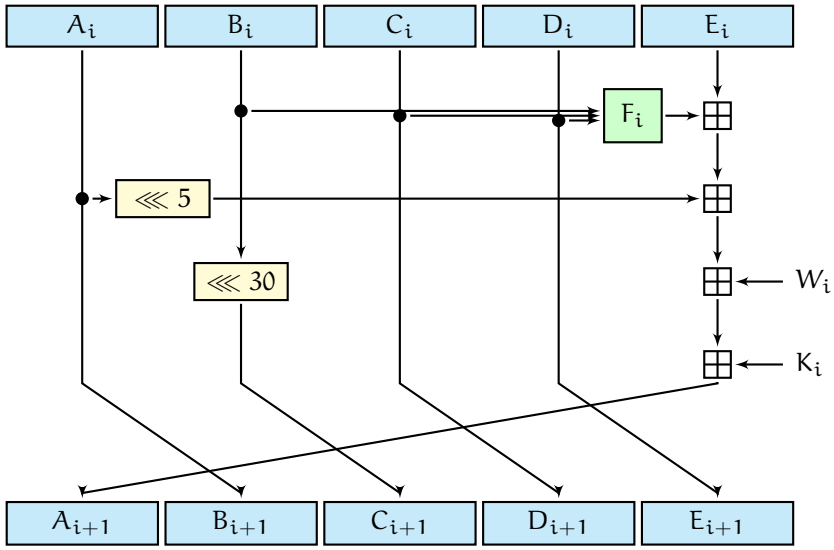


Figure 10: Outline [35] of one-round (out of 80) of the SHA-1 compression function. The function uses modular addition $\boxplus$, bit-wise rotation $\lll$ and a boolean function F_i. The expanded message words are W_i and K_i is a round specific constant.

.

of operation building a function which takes arbitrary sized input and generates arbitrary sized output. It is based on a fixed-sized permutation π, a padding rule and takes two parameters: the rate r and the capacity c. The sponge construction is also iterative and operates on an internal state S of size $b = r + c$.

First, split the input M into blocks $M_1, \ldots, M_n$ of size r and apply the padding. Set the initial state $S = (0 \ldots 0)$ and process in the following two steps:

- **Absorb:** XOR the ith message block to the first r bits of the state and update it with the π-permutation. Repeat this step until all message blocks have been processed.
- **Squeeze:** Append the first r bits of the state to h and update the state. Repeat this step until enough output has been generated.

An outline of this procedure can be seen in Figure 11. When using a random permutation the sponge construction is as secure as a random oracle apart from inner collisions [8].

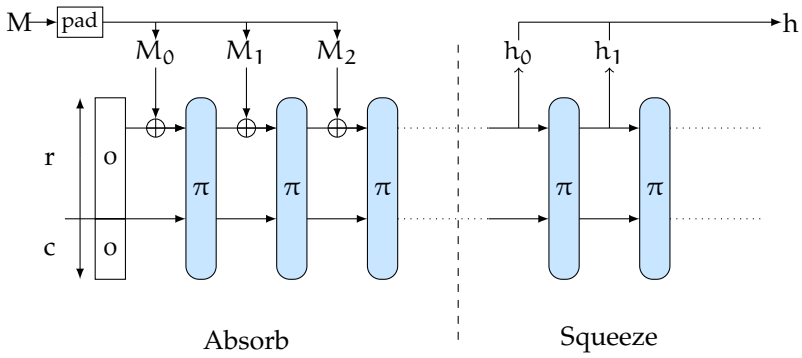


Figure 11: The sponge construction takes input of arbitrary length and computes an output of arbitrary length. It uses a fix-sized permutation π and the input is processed iteratively.

As the output size of a sponge can be arbitrary, it is not possible to define the security in terms of the output size like we did previously for hash functions. If this was the case one could simply produce more output in the squeeze phase to increase the level of security. The security of the sponge construction is strongly tied to the security parameter c. A sponge with a capacity of c provides $2^{c/2}$ collision and $2^{c/2}$ (second) preimage resistance. The reason for the reduced costs of a preimage attack are that an attacker can apply a meet-in-the-middle attack (see Section 4.1.2).

3.4 Hash-based Signature Schemes

Digital signature schemes are an important cryptographic application providing a digital equivalent to *handwritten* signatures. A signature is a bit string which depends on a secret exclusively known to the signer (secret key) and on the content of the message which is signed. The validity of the signature can be verified by any public party without knowledge of this secret key through some additional information (public key).

Let $\mathcal{M}$ be the message space. A digital signature scheme is a triple of efficiently computable algorithms $(\mathbf{Kg}, \mathbf{Sign}, \mathbf{Vf})$:

- **Key Generation ($\mathbf{G}(1^n)$):** Given a security parameter 1^n, generates both the secret key (sk) which is required to sign messages and the public key (pk) to verify the signatures.
- **Signing ($\mathbf{S}(M, sk)$):** Takes as input a message $M \in \mathcal{M}$ and sk to produce a digital signature σ.
- **Verify ($\mathbf{V}(M, pk, \sigma)$):** Returns 1 if the signature σ is valid for M given the public key pk.

The most commonly used signature schemes in practice are RSA [61], DSA [27] and ECDSA. These schemes are not secure in the setting of quantum computing as they are based on the difficulty of factoring large integers or computing discrete logarithms. Both these problems can be efficiently solved on a quantum computer [68].

Hash-based signatures seem to be a promising alternative to these schemes as they only rely on the security of the underlying hash-function used. While there is no proof that they are quantum resistant they only require minimal security assumptions, like a collision resistant hash function. In fact one-way functions are necessary for a secure signature scheme to exist [63].

3.4.1 Security Goals

Similar to a handwritten signature it should be difficult for an adversary to *forge* a signature which appears to be valid. The worst case is if an adversary is able to recover the secret key or construct an efficient algorithm which is functionally equivalent to the signature algorithm instantiated with the secret key. This would allow the adversary to forge arbitrary signatures.

A weaker attack scenario is if an adversary is able to produce *selective forgeries*. This means an attacker is able to construct valid signatures for a specific message or class of messages. As an example, consider an adversary observing some valid signatures and then use this information to derive a valid signature for a new message.

3.4.2 One-time digital signatures

The basic building block for a hash-based signature scheme are one-time digital signatures. As the name suggest, these schemes only allow to sign a single message with a given key pair.

Lamport-Diffie. A classic example for such a scheme is the *Lamport-Diffie one-time signature scheme* (LD-OTS). It requires a one-way function

$$\mathcal{F} : \{0,1\}^n \to \{0,1\}^n$$

and a cryptographic hash function

$$\mathcal{H} : \{0,1\}^* \to \{0,1\}^n.$$

Key Generation. The secret key consists of $2n$ bit strings $x_i[j]$ of length n which are uniformly chosen at random

$$sk = (sk_0[0], sk_0[1], \ldots, sk_{n-1}[0], sk_{n-1}[1]).$$

The public key is then given by

$$pk = (pk_0[0], pk_0[1], \ldots, pk_{n-1}[0], pk_{n-1}[1])$$

where

$$pk_i[j] = \mathcal{F}(sk_i[j]), \qquad i \in \{0, \ldots, n-1\}, j \in \{0,1\}.$$

The resulting key pair (sk, pk) is then of size $2(2n^2)$.

Signing. For signing an arbitrary sized message M we first compute the hash of the the message

$$h = \mathcal{H}(M) = (h_0, \ldots, h_{n-1})$$

and then compute the signature in the following way

$$\sigma = (sk_0[h_0], sk_1[h_1], \ldots, sk_{n-1}[h_{n-1}]).$$

The length of the signature is therefore n^2.

Verify. The verifier computes $h = \mathcal{H}(M)$ again and then checks if

$$f(\sigma_i) = pk_i[h_i] \qquad i \in \{0, \ldots, n-1\}.$$

The key pair can only be used once without compromising security, as the signature contains half of the secret key. As an example assume an adversary observes two messages M, M', with $\mathcal{H}(M) = (1,1,0,0)$ and $\mathcal{H}(M') = (1,1,1,1)$, being signed. The signatures are $\sigma = (sk_0[1], sk_1[1], sk_2[0], sk_3[0])$ respectively $\sigma' = (sk_0[1], sk_1[1], sk_2[0], sk_3[1])$. The information leaked on the secret key would clearly allow an adversary to construct a valid signature for $\mathcal{H}(M) = (1,1,1,0)$ and $\mathcal{H}(M) = (1,1,0,1)$ as she knows the necessary parts of the secret key.

Winternitz One of the major drawbacks of LD-OTS is the rather large size of the signature. The Winternitz OTS (W-OTS), first mentioned in [51], improves upon this by using the parts of the secret key to sign more than a single bit. These scheme has been further improved by Hülsing [34], named W-OTS+, which both provides shorter signature and allows to drop the requirement for collision resistance of the hash function.

3.4.3 Merkle Signature Scheme

The requirement of generating a new key pair for each signatures makes one-time signatures inefficient in practice. Ralph Merkle proposed a solution for this problem based on binary hash-trees in 1979, the Merkle signature scheme (MSS) [51]. The main advantage of this scheme is that it bundles a number of one-time signature and allows to use a single smaller public key to use for verification.

Key Generation. The signer first has to select a parameter N, which specifies the number of messages 2^N which can be signed with this key pair. In the next step the signer generates 2^N one-time signature key pairs (sk_i, pk_i). The public keys pk_i are used to generate the leaves of a binary tree by computing $\mathcal{H}(pk_i)$ (see Figure 12). In the binary tree a parent node always contains the hash of the concatenation of its children. The key pair is then given by

$$(sk, pk) = ((sk_0, sk_1, \ldots sk_{N-1}), pk).$$

Signing. The signer picks an index i and signs the message using the one-time signature to get σ_{OTS}. Note that the signer has to keep track of not reusing the index i and therefore MSS is a so-called *stateful* scheme. Additionally, the signature has to contain additional information, the authentication path $A_i = (a_0, a_1, \ldots)$, to allow a verifier to check whether the provided signature σ_{OTS} is indeed part of the hash tree

$$\sigma = (i, \sigma_{OTS}, pk_i, A_i).$$

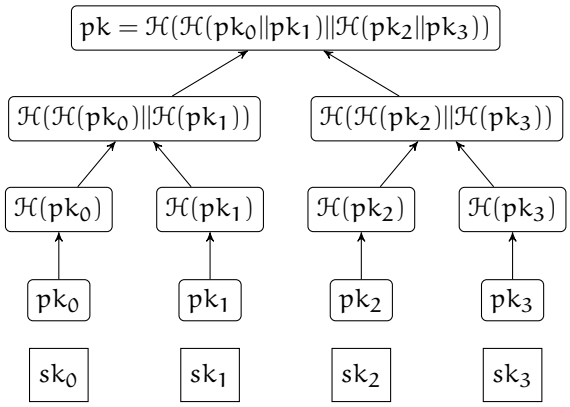


Figure 12: Key generation in the Merkle Signature Scheme.

Verify. The verification is done in two steps. First, verify that the one-time signature σ_{OTS} is valid. The next step is to check the authenticity of pk_i. The auxiliary information given in the signature allows the verifier to recompute the root of the tree and verify that the used OTS was indeed part of this tree. As an example see Figure 13.

3.4.4 XMSS

XMSS (eXtended Merkle Signature Scheme) [15] is a hash-based digital signature scheme improving upon the classic version of MSS in various aspects. XMSS provides forward security, this means that if a key gets compromised all previously generated signatures still remain valid. Another advantage of XMSS is that it only requires two function families $\mathcal{H}$ and $\mathcal{F}$, where $\mathcal{H}$ has to be second preimage resistant and $\mathcal{F}$ pseudo random.

3.4.5 Stateless Schemes

Using a one-time signature twice breaks the security of the system. All the previous signature schemes share the property of being stateful. This means that the signer has to keep track of which one-time signatures have been used and never use them again. This might be acceptable for some applications but can also be a serious drawback in practice. For instance if you want to backup your key or move it to another computer you have to make sure that the *state* is synchronized.

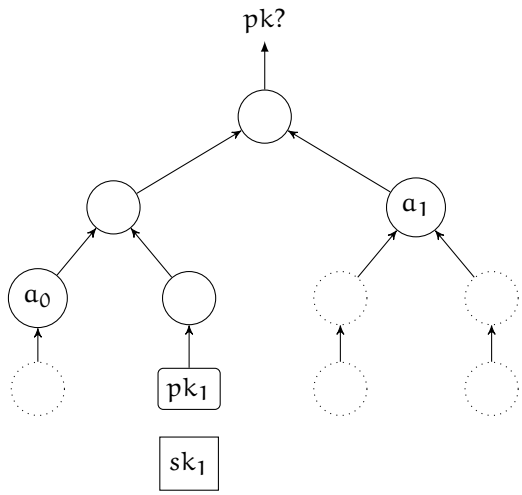


Figure 13: Verification of a signature in the Merkle signature scheme for $i = 1$. Using a_0, a_1 the verifier can compute the root of the tree.

Goldreich. Goldreich proposed a *stateless* signature scheme in [28, 29], which uses a binary tree build out of one-time signature key pairs. The key pairs corresponding to the leafs are used to sign messages, while the key pairs corresponding to the non-leafs are used to sign the public keys of its child nodes. Similar to the MSS the root node becomes the public key.

One of the main disadvantages of this schemes is the signature size. The signature contains all the public keys in the path from the leaf to the root, all the public keys of the siblings, the OTS on the message and the OTS on all the public keys in the path. This leads to a signature size which is cubic in the security parameter. For a typical security parameter $n = 256$, the scheme by Goldreich would have a signature size above 1 MB.

SPHINCS. SPHINCS [7] is a fairly new hash-based signature scheme which is stateless and provides much shorter signatures then the scheme by Goldreich. The main improvements in SPHINCS which allow reducing the signature size are the use of a *few-time* signature scheme and the use of a hyper-tree construction which allows a trade-off between signature size and computation time. Together this leads to a signature of $\approx$ 41KB for a security level of 128-bit against an attacker with a quantum computer.

4

Cryptanalysis

While cryptography is about designing new algorithms and applications which are secure, *cryptanalysis* is concerned with evaluating the security of these algorithms and tries to *break* these algorithms. This usually means to subvert the confidentiality, integrity or authenticity provided. Cryptography and cryptanalysis are very closely connected, as when designing a new algorithm one has to take into account cryptanalytic techniques to obtain a secure design.

In the following we will look at some of the most powerful techniques known today which every newly designed cryptographic algorithm should resist.

4.1 Meet-in-the-middle

The meet-in-the-middle attack is a generic attack applicable to a large variety of cryptographic primitives. The main idea is to split the primitive into two independent parts and use a time-memory trade-off for a more efficient attack. While the original meet-in-the-middle attack is very generic, many improvements to it have been suggest by using the underlying structure of the cryptographic primitive.

One of the first applications of this technique is to multiple encryption [25], which we will have a closer look at in Section 4.1.1. Since then many improvements have been made: Better trade-offs [53], the first attack on full round AES using the bi-clique technique [12] and methods to automatically find meet-in-the-middle attacks [22]. For cryptographic hash function meet-in-the-middle attacks are mainly used for finding preimages, for example MD4 [2, 45], MD5 [64], SHA-1 [3] and SHA-2 [1, 39].

4.1.1 Multiple Encryption

One of the most prominent examples of meet-in-the-middle attacks is the application to *multiple encryption*. As the key size of DES is limited to 56-bit, it was suggested to encrypt two or three times with different keys resulting in a key size of 112 respectively 168 bits. However, by using a meet-in-the-

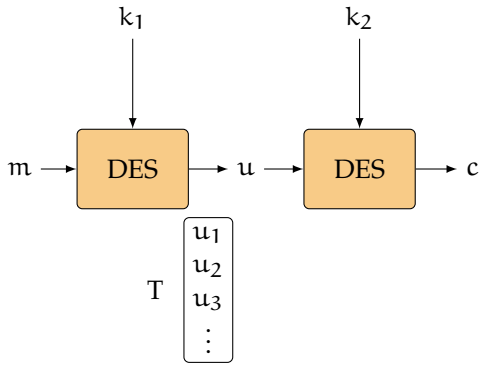


Figure 14: Outline of the meet-in-the-middle attack on double-DES.

middle attack we can show that the effective key size is significant smaller. As an example we look at DES using double encryption

$$\text{doubleDES}(m) = \text{DES}_{k_2}(\text{DES}_{k_1}(m)). \tag{16}$$

We can recover the two keys k_1, k_2 in the following way:

1. Obtain one plaintext/ciphertext pairs (m, c).
2. Compute the encryption of $u = \text{DES}_{k_1}(m)$ under all possible keys k_1 and store them in a table T.
3. Decrypt the ciphertext $v = \text{DES}_{k_2}(c)$ for all possible keys k_2 and check if $v \in T$. If we find such a v then (k_1, k_2) is a candidate for the correct key.

Complexity. DES has a 64-bit block size and 56-bit key size, therefore step (2) costs 2^{56} time and requires 64×2^{56} bits of memory. Step (3) again requires to do 2^{56} DES encryptions and for each one check whether the entry is in T. At this point we could still have gotten a wrong key pair as the expected number of key pairs for our plaintext/ciphertext pair is $2^{56}2^{56}/2^{64} = 2^{48}$. However, if we use a second plaintext/ciphertext pair then the probability that a wrong key pair also gives a correct solution is only $2^{48}/2^{64} = 2^{-16}$, allowing us to filter out any false guesses with a high probability.

Oorschot and Wiener [53] showed that the product of time and memory complex can be kept constant, which allows us more efficient trade-offs as the memory complexity is often more critical in practice.

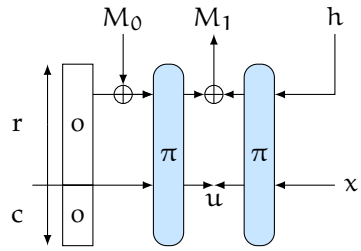


Figure 15: Finding a preimage for a Sponge using an inner collision.

4.1.2 Preimages in a Sponge Construction

The reason why we only get $2^{c/2}$ preimage resistance in the Sponge construction is that an attacker can find a collision on the inner state to construct a preimage. Assume we build a hash function with output size n using the Sponge construction with capacity c and rate r. We can find a preimage for a given h by computing $\pi^{-1}(h \parallel x)$, for random values of x, and choosing a random message block M_0 to compute $\pi(M_0 \parallel 0)$ and check if we have a collision on the last c bits.

This means that if we want to construct a hash function using the sponge construction, the capacity should be $c = 2n$ to have the same security level as one would expect from a hash function with n-bit output size.

4.2 Differential Cryptanalysis

Differential cryptanalysis is one of the most powerful techniques and is widely used for analyzing both block ciphers and cryptographic hash functions. It was first published by Biham and Shamir [10] to analyze the DES block cipher, which surprisingly turned out to be fairly resistant against this type of attack and highly suggests that it was taken into account by the designers of DES.

The basic idea behind differential cryptanalysis is to find a correlation between the difference of two plaintexts and the corresponding ciphertexts (see Figure 16). If such a correlation occurs with a significant higher probability in a block cipher then it does for a random permutation we can use this fact for an attack.

The difference is chosen to not be influenced by the addition of the secret key, which usually corresponds to addition of vectors over $\mathbb{F}_2$ (XOR) or addition in $\mathbb{Z}_n$. In the following we assume that the key is added using the XOR operation and $f : \mathbb{F}_2^n \to \mathbb{F}_2^n$.

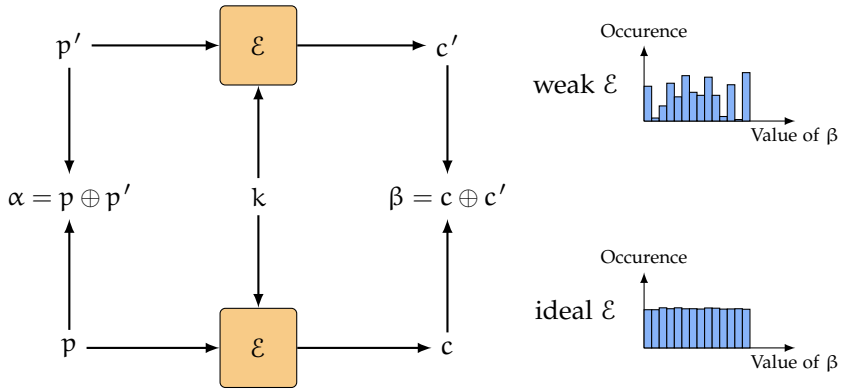


Figure 16: Differential cryptanalysis observes the correlation between the difference of a pair of plaintexts (p, p') and the corresponding ciphertexts (c, c'). For an ideal block cipher we expect all differences β to be close to some probability distribution [20].

Definition 4.1. A *differential* is a pair of differences $(\alpha, \beta) \in \mathbb{F}_2^n \times \mathbb{F}_2^n$, where α is the *input difference* to some function f and β the *output difference*.

We use $\alpha \xrightarrow{f} \beta$ to denote a differential for a function f. We denote a pair of messages $(x, x \oplus \alpha)$ and $f(x) \oplus f(x \oplus \alpha) = \beta$ as $\alpha \xrightarrow[x]{f} \beta$. We call this a *right pair* or a pair which follows the differential $\alpha \xrightarrow{f} \beta$.

In a block cipher we usual don't know the input to the function f, as a secret uniformly at random chosen key is masking it. Hence, for any non-linear function f the transition $\alpha \xrightarrow{f} \beta$ will be probabilistic.

Definition 4.2. The *differential probability* of a differential $\alpha \xrightarrow{f} \beta$ is given by

$$\mathrm{DP}(\alpha \xrightarrow{f} \beta) = \Pr_X(f(X) \oplus f(X \oplus \alpha) = \beta). \tag{17}$$

For a random function the probability for any non-zero differential is close to 2^{-n}. Therefore, if we can find a differential with a higher probability for our function f, we can distinguish it from a random function. In general it is difficult to compute the DP for a differential exactly and for the usual block sizes, i.e. 64 or 128 bits, it is infeasible to compute it by trying all possible inputs.

In order to get a good approximation of the actual DP with less computational effort we can use the underlying structure of f. Most block ciphers used in practice are iterative block ciphers (see Theorem 2.2) and we can trail how the differences propagate through each round.

Definition 4.3. A *differential trail*[1] Q is a sequence of differences

$$Q = (\alpha_0 \xrightarrow{f_0} \alpha_1 \xrightarrow{f_1} \cdots \alpha_{r-1} \xrightarrow{f_{r-1}} \alpha_r). \tag{18}$$

The first step is to determine the probability of a differential for a single round and later connect them together to cover more rounds. Finding a good or the best differential for a single round can usually be done in practice.

In the case of SPNs which use 4-bit or 8-bit S-boxes, we can actually compute all differentials for a single S-box and store them in a *differential distribution table* (DDT), allowing us to look up the differential with the highest probability.

At this point we can compute the probability for one round, however it is still open how the probability can be computed when we connect multiple rounds

$$DP(Q) = Pr(\alpha_0 \xrightarrow{f_0} \alpha_1 \xrightarrow{f_1} \cdots \alpha_{r-1} \xrightarrow{f_{r-1}} \alpha_r). \tag{19}$$

In general it is not feasible to determine the exact value of Equation 19 and the common approach is to make two assumptions. The first assumption we make is that the rounds are independent, which allows us to simplify Equation 19 to

$$DP(Q) = \prod_{i=0}^{r-1} DP(\alpha_i \xrightarrow{f_i} \alpha_{i+1}). \tag{20}$$

While this assumption of independent rounds is not true in general it serves as a good approximation in practice.

Another problem which occurs is that the attacker only receives the messages encrypted under a *fixed* key. Therefore, in practice we want to determine the probability of the differential for many or all keys. This is however not possible in most cases, hence we assume that the probability for a fixed key is close to the probability for a random key.

Definition 4.4. The *Stochastic Equivalence Hypothesis* states that for all high probability differentials $(\alpha \xrightarrow{f} \beta)$

$$\Pr_{\mathcal{M}}(\alpha \xrightarrow[\mathcal{M}]{\mathcal{E}_K} \beta) \approx \Pr_{\mathcal{M},\mathcal{K}}(\alpha \xrightarrow[\mathcal{M}]{\mathcal{E}_{\mathcal{K}}} \beta) \tag{21}$$

holds for a large fraction of the keys K.

In practice, we can not observe the exact differences in all the intermediate states, but only observe the output difference between the ciphertext pair. Therefore for an attack we are actually interested in the probability of

[1] Another common term for this is *differential characteristic* or *differential path*.

the differential $\alpha_0 \xrightarrow{f} \alpha_r$. We can compute this probability by summing the probability of all trails with the same input and output difference

$$\Pr(\alpha_0 \xrightarrow{f} \alpha_r) = \sum_{\alpha_1,\ldots,\alpha_{r-1}} \Pr(\alpha_0 \xrightarrow{f_0} \alpha_1 \xrightarrow{f_1} \cdots \alpha_{r-1} \xrightarrow{f_{r-1}} \alpha_r). \tag{22}$$

If an attacker can now find such a differential with a probability $> 2^{-n}$ it can be used to recover information of the secret key.

4.2.1 Key Recovery using a Differential Distinguisher

In the following we will look at how we can use differential cryptanalysis to recover information of the secret key in a block cipher. In our example we use the block cipher depicted in Figure 17 with block size n, r rounds and each sub-key $k_0, \ldots, k_r$ being k-bit.

The first step would be to find a differential $Q = \alpha \to \beta$ over $r-1$ rounds with a probability of p. Then we can proceed in the following way to recover the correct last round key:

1. We request the encryption of N messages m of the form $(m, m \oplus \alpha)$ and obtain the corresponding ciphertext pairs (c, c').
2. Initialize a set of counters $T_0, \ldots, T_{k-1}$, one for each possible last round key, and set them to zero.
3. For each pair of ciphertexts (c, c') obtained:
   - For each possible choice of the last round key k_r:
     - Compute the intermediate values $v = f^{-1}(c \oplus k_r)$ resp. $v' = f^{-1}(c' \oplus k_r)$. This allows us to determine $v \oplus v' = u \oplus u'$.
     - If $u \oplus u' = \beta$ increment the counter T_{k_r}.
4. The counter with the value Np is most likely the correct key.

To understand why this attack works and what the success probability is we have to consider when we increase the counters in step (3). When decrypting one round with a wrong key we can still get the correct difference β for some plaintext pairs, suggesting that this could be a potential key candidate. However, if we have a right pair which follows the differential then the correct key will always be amongst the possible key candidates.

To make the attack work, we require that the correct key lies more often amongst the key candidates then a wrong key. We need at least one pair of plaintexts following the differential, therefore $N = tp^{-1}$ for some constant $t > 0$. Additionally, we assume that for a wrong key guess the set of key

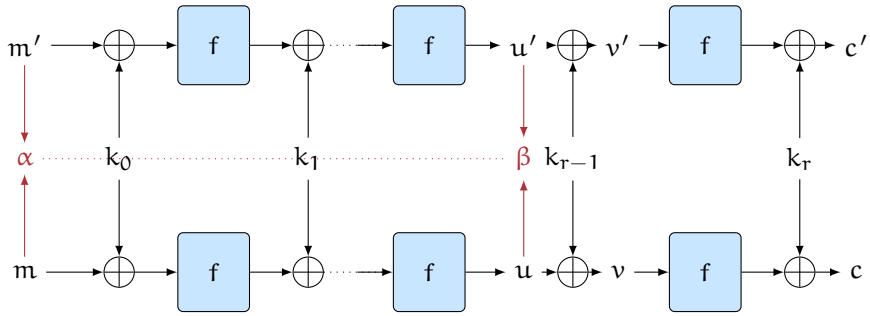


Figure 17: Using a differential distinguisher to recover the key.

candidates is randomly distributed. Intuitively what happens when we guess the wrong key is that we encrypt for an additional round making the output more *random*.

In some cases it is already possible to *filter* the ciphertext pairs before the key guessing part, which can further help reducing the noise. For instance, in a Feistel network we could observe one half of the state.

4.2.2 Truncated Differentials

The idea behind *truncated differentials* [40] is that it is not always necessary to predict all bits of a differential and it can be beneficial to only consider parts.

Definition 4.5. A *truncated difference* α is a n-bit value $\alpha \in \{0, 1, \star\}^n$, where 1 corresponds to a difference in this bit, 0 to no difference and $\star$ that this bit is unknown.

We can then in a similar way define differentials

Definition 4.6. A *truncated differential* is a pair of truncated differences (α, β).

One of the main advantages of truncated differentials is that they allow us to bundle a larger number of differentials together and therefore can have a significant higher probability. Additionally, it is often sufficient to only know parts of the difference to propagate them through the round functions. Especially for SPN constructions this approach can be very useful, when considering only whether an S-box has a difference (is active) or no difference (not active).

4.2.3 Impossible Differentials

The goal for an attacker is typically to find differentials with high probability, however we can also utilize if a block cipher has differentials with a prob-

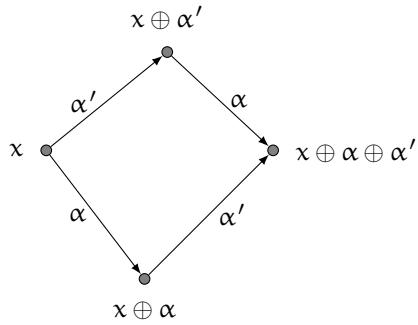


Figure 18: Using a structure we can obtain two pairs for each differential using only four plaintexts.

ability of zero [9, 41]. If such an differential exists we can use it to build a distinguisher similar to the classic differential attack.

The simplest way to construct such an *impossible differential* is to use two differentials (α, β) and (γ, δ) where the former covers the first part of the cipher and the other one the second part. We also require that both differentials have a probability of 1. If $\beta \neq \gamma$, then the differential (α, δ) will have a probability of zero. Finding such an impossible differential is often done using truncated differentials.

4.2.4 Structures

Using multiple differentials in attack can be beneficial by increasing the success rate or allowing to recover different parts of the key and lead to more efficient attacks. However, one of the main drawbacks, and often limiting factors of differential cryptanalysis, is the large amount of data required.

In a differential attack an attacker would ask for the encryption of random plaintexts x and $x \oplus \alpha$ in order to obtain one pair of plaintexts with the correct difference. Consider now that we use a second differential and ask for the encryption of x, $x \oplus \alpha$, $x \oplus \alpha'$ and $x \oplus \alpha \oplus \alpha'$. By combining these plaintexts we can now obtain two pairs for each difference α and α' (see Figure 18). In general if we have d differentials we can build a structure out of 2^d plaintexts giving us 2^{d-1} pairs for each differential.

4.2.5 Collision Attacks for Hash Functions

Differential cryptanalysis also plays an important role in the analysis of hash functions especially when trying to find collisions. This approach seems very intuitive for finding collisions, as one wants to find a pair of messages

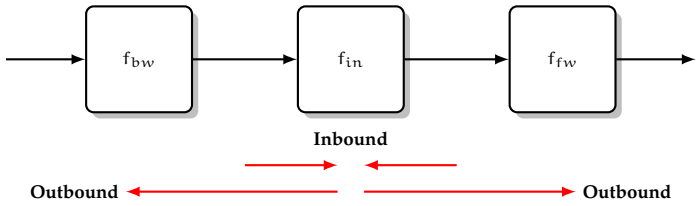


Figure 19: Outline of the rebound attack.

(m, m') with non zero difference α giving $\mathcal{H}(m) = \mathcal{H}(m')$. We are therefore interested in differentials of the form $\alpha \xrightarrow{\mathcal{H}} 0$.

For an efficient attack we also want a differential with high probability, similar to the block cipher case. However, finding a pair of messages which follow the differential is a very distinct process. In a block cipher we usually require to query 2^{-p} random message pairs, where p is the probability of the differential, to expect one right pair. This is due to the addition of the secret key, which masks the input.

For hash functions we do not use any secret key and an attacker has full control over the input to the hash function. This allows an attacker to choose the right pair which follows the differential and also verify intermediate states. Using this available degrees of the freedom the probability of the differential does not matter, as long as the attacker can freely choose and modify the message. In a good hash function each output bit should depend on each input bit after a few rounds limiting this approach as it quickly requires solving highly non-linear equations to find a message following the differential over many rounds. Nonetheless, using these so-called *message modification techniques* has lead to the break of widely used hash functions like MD5 [76] or SHA-1 [75].

4.2.6 Rebound Attack

The *rebound attack* is a powerful tool in the cryptanalysis of hash functions and can be seen as a message modification technique. It is especially useful for finding collisions for AES-based designs [44, 49]. The basic idea of the rebound attack is to split the hash function into three sub-functions (see Figure 19) and proceed in two steps. First, the inbound phase which tries to find an efficient solution for the middle part using the available degrees of freedom. This is followed by a probabilistic part, the outbound phase in f_{fw} and f_{bw} using the solutions from the inbound phase.

The rebound attack uses truncated differentials, which are constructed such that the most expensive part of the trail is covered by f_{in} and the part of the trail in f_{fw} and f_{bw} holds with high probability.

Bibliography

[1] Kazumaro Aoki, Jian Guo, Krystian Matusiewicz, Yu Sasaki, and Lei Wang. "Preimages for Step-Reduced SHA-2." In: *Advances in Cryptology - ASIACRYPT 2009*. 2009, pp. 578–597.

[2] Kazumaro Aoki and Yu Sasaki. "Preimage Attacks on One-Block MD4, 63-Step MD5 and More." In: *Selected Areas in Cryptography, 15th International Workshop, SAC 2008*. 2008, pp. 103–119.

[3] Kazumaro Aoki and Yu Sasaki. "Meet-in-the-Middle Preimage Attacks Against Reduced SHA-0 and SHA-1." In: *Advances in Cryptology - CRYPTO 2009*. 2009, pp. 70–89.

[4] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. *The SIMON and SPECK Families of Lightweight Block Ciphers*. Cryptology ePrint Archive, Report 2013/404. http://eprint.iacr.org/. 2013.

[5] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. "The SKINNY Family of Block Ciphers and its Low-Latency Variant MANTIS." In: *Advances in Cryptology - CRYPTO 2016*. 2016.

[6] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh V. Vazirani. "Strengths and Weaknesses of Quantum Computing." In: *SIAM J. Comput.* 26.5 (1997), pp. 1510–1523.

[7] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. "SPHINCS: Practical Stateless Hash-Based Signatures." In: *Advances in Cryptology - EUROCRYPT 2015*. 2015, pp. 368–397.

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *Sponge functions*. Ecrypt Hash Workshop 2007. 2007.

[9] Eli Biham, Alex Biryukov, and Adi Shamir. "Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials." In: *Advances in Cryptology - EUROCRYPT '99*. 1999, pp. 12–23.

[10] Eli Biham and Adi Shamir. "Differential Cryptanalysis of DES-like Cryptosystems." In: *Advances in Cryptology - CRYPTO '90*. 1990, pp. 2–21.

[11] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications." In: *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. 2016, pp. 292–302.

[12] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. "Biclique Cryptanalysis of the Full AES." In: *Advances in Cryptology - ASIACRYPT 2011*. 2011, pp. 344–371.

[13] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. "PRESENT: An Ultra-Lightweight Block Cipher." In: *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*. 2007, pp. 450–466.

[14] Julia Borghoff et al. "PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract." In: *Advances in Cryptology - ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Springer.

[15] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. "XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions." In: *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*. 2011, pp. 117–129.

[16] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M Matyas Jr, Luke O'Connor, Mohammad Peyravian, David Safford, et al. "MARS-a candidate cipher for AES." In: *NIST AES Proposal* 268 (1998).

[17] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. "Merkle-Damgård Revisited: How to Construct a Hash Function." In: *Advances in Cryptology - CRYPTO 2005*. 2005, pp. 430–448.

[18] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. *The NOEKEON Block Cipher*. Submission to the NESSIE project. 2000.

[19] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[20] Joan Daemen and Vincent Rijmen. "Probability distributions of correlation and differentials in block ciphers." In: *J. Mathematical Cryptology* 1.3 (2007), pp. 221–242.

[21] Ivan Damgård. "A Design Principle for Hash Functions." In: *Advances in Cryptology - CRYPTO '89*. 1989, pp. 416–427.

[22] Patrick Derbez and Pierre-Alain Fouque. "Automatic Search of Meet-in-the-Middle and Impossible Differential Attacks." In: *Advances in Cryptology - CRYPTO 2016*. 2016, pp. 157–184.

[23] Tim Dierks. *The transport layer security (TLS) protocol version 1.2*. RFC 5246. RFC Editor, 2008. URL: https://tools.ietf.org/html/rfc5246.

[24] Whitfield Diffie and Martin E. Hellman. "New directions in cryptography." In: *IEEE Trans. Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638. URL: http://dx.doi.org/10.1109/TIT.1976.1055638.

[25] Whitfield Diffie and Martin E. Hellman. "Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard." In: *IEEE Computer* 10.6 (1977), pp. 74–84.

[26] Naganand Doraswamy and Dan Harkins. *IPSec: the new security standard for the Internet, intranets, and virtual private networks*. Prentice Hall, 2003.

[27] Taher El Gamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms." In: *Advances in Cryptology, CRYPTO '84*. 1984, pp. 10–18.

[28] Oded Goldreich. "Two Remarks Concerning the Goldwasser-Micali-Rivest Signature Scheme." In: *Advances in Cryptology - CRYPTO '86*. 1986, pp. 104–110.

[29] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004. ISBN: 0-521-83084-2.

[30] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract)." In: *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*. 1985, pp. 291–304.

[31] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search." In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*. 1996, pp. 212–219.

[32] Martin E. Hellman. "A cryptanalytic time-memory trade-off." In: *IEEE Trans. Information Theory* 26.4 (1980), pp. 401–406.

[33] Shoichi Hirose. "Provably Secure Double-Block-Length Hash Functions in a Black-Box Model." In: *Information Security and Cryptology - ICISC 2004*. 2004, pp. 330–342.

[34] Andreas Hülsing. "W-OTS+ - Shorter Signatures for Hash-Based Signature Schemes." In: *Progress in Cryptology - AFRICACRYPT 2013*. 2013, pp. 173–188.

[35] Jérémy Jean. *TikZ for Cryptographers*. http://www.iacr.org/authors/tikz/. 2016.

[36] Antoine Joux. "Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions." In: *Advances in Cryptology - CRYPTO 2004*. 2004, pp. 306–316.

[37] D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996. ISBN: 9781439103555.

[38] John Kelsey and Bruce Schneier. "Second Preimages on n-bit Hash Functions for Much Less than 2^n Work." In: *IACR Cryptology ePrint Archive* 2004 (2004), p. 304.

[39] Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva. "Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family." In: *Fast Software Encryption - 19th International Workshop, FSE 2012*. 2012, pp. 244–263.

[40] Lars R. Knudsen. "Truncated and Higher Order Differentials." In: *Fast Software Encryption, 2nd International Workshop, FSE 1994*. 1994, pp. 196–211.

[41] Lars R. Knudsen. *DEAL - A 128-bit Block Cipher*. 1998.

[42] Lars R. Knudsen and Matthew Robshaw. *The Block Cipher Companion*. Information Security and Cryptography. Springer, 2011.

[43] Xuejia Lai. "On the design and security of block ciphers." PhD thesis. Swiss Federal Institute of Techology, 1992.

[44] Mario Lamberger, Florian Mendel, Martin Schläffer, Christian Rechberger, and Vincent Rijmen. "The Rebound Attack and Subspace Distinguishers: Application to Whirlpool." English. In: *Journal of Cryptology* (2013), pp. 1–40. ISSN: 0933-2790. DOI: 10.1007/s00145-013-9166-5. URL: http://dx.doi.org/10.1007/s00145-013-9166-5.

[45] Gaëtan Leurent. "MD4 is Not One-Way." In: *Fast Software Encryption, 15th International Workshop, FSE 2008*. 2008, pp. 412–428.

[46] Moses Liskov, Ronald L. Rivest, and David Wagner. "Tweakable Block Ciphers." In: *Advances in Cryptology - CRYPTO 2002*. 2002, pp. 31–46.

[47] Michael Luby and Charles Rackoff. "How to Construct Pseudorandom Permutations from Pseudorandom Functions." In: *SIAM J. Comput.* 17.2 (1988), pp. 373–386.

[48] Mitsuru Matsui. "New Block Encryption Algorithm MISTY." In: *Fast Software Encryption, FSE '97*. 1997, pp. 54–68.

[49] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. "The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl." In: *Fast Software Encryption, 16th International Workshop, FSE 2009*. 2009, pp. 260–276.

[50] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[51] Ralph C. Merkle. "A Certified Digital Signature." In: *Advances in Cryptology - CRYPTO '89*. 1989, pp. 218–238.

[52] Ralph C. Merkle. "One Way Hash Functions and DES." In: *Advances in Cryptology - CRYPTO '89*. 1989, pp. 428–446.

[53] Paul C. van Oorschot and Michael J. Wiener. "Improving Implementable Meet-in-the-Middle Attacks by Orders of Magnitude." In: *Advances in Cryptology - CRYPTO '96*. 1996, pp. 229–236.

[54] Colin Percival. "Stronger key derivation via sequential memory-hard functions." In: *BSDCan 2009* (2009).

[55] Niels Provos and David Mazieres. "A future-adaptable password scheme." In: *Proceedings of the Annual USENIX Technical Conference*. 1999.

[56] Jean-Jacques Quisquater and Jean-Paul Delescaille. "How Easy is Collision Search. New Results and Applications to DES." In: *Advances in Cryptology - CRYPTO '89*. 1989, pp. 408–413.

[57] Michael O. Rabin. "Digitalized signatures." In: *Foundations of Secure Computations*. 1978, pp. 155–166.

[58] Ronald L. Rivest. "The MD4 Message Digest Algorithm." In: *Advances in Cryptology - CRYPTO '90*. 1990, pp. 303–311.

[59] Ronald L. Rivest. *The MD5 Message Digest Algorithm*. RFC 1321 (Informational). Internet Engineering Task Force, 1992.

[60] Ronald L Rivest, MJB Robshaw, Ray Sidney, and Yiqun Lisa Yin. "The RC6TM block cipher." In: *First Advanced Encryption Standard (AES) Conference*. 1998.

[61] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Commun. ACM* 21.2 (1978), pp. 120–126.

[62] Phillip Rogaway. "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC." In: *Advances in Cryptology - ASIACRYPT 2004*. 2004, pp. 16–31.

[63] John Rompel. "One-Way Functions are Necessary and Sufficient for Secure Signatures." In: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*. 1990, pp. 387–394.

[64] Yu Sasaki and Kazumaro Aoki. "Finding Preimages in Full MD5 Faster Than Exhaustive Search." In: *Advances in Cryptology - EUROCRYPT 2009*. 2009, pp. 134–152.

[65] Bruce Schneier and John Kelsey. "Unbalanced Feistel Networks and Block Cipher Design." In: *Fast Software Encryption, FSE '96*. 1996, pp. 121–144.

[66] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. "Twofish: A 128-bit block cipher." In: *NIST AES Proposal* 15 (1998).

[67] Claude E Shannon. "Communication theory of secrecy systems." In: *Bell system technical journal* 28.4 (1949), pp. 656–715.

[68] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." In: *SIAM J. Comput.* 26.5 (1997), pp. 1484–1509.

[69] National Institute of Standards and Technology. *Data Encryption Standard (DES)*. 1977.

[70] National Institute of Standards and Technology. *Secure Hash Standard*. 1993.

[71] National Institute of Standards and Technology. *Secure Hash Standard*. 1995.

[72] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*. 2001.

[73] Serge Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." In: *Advances in Cryptology - EUROCRYPT 2002*. 2002, pp. 534–546.

[74] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. "Cryptanalysis of the Hash Functions MD4 and RIPEMD." In: *Advances in Cryptology - EUROCRYPT 2005*. 2005, pp. 1–18.

[75] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. "Finding Collisions in the Full SHA-1." In: *Advances in Cryptology - CRYPTO 2005*. 2005, pp. 17–36.

[76] Xiaoyun Wang and Hongbo Yu. "How to Break MD5 and Other Hash Functions." In: *Advances in Cryptology - EUROCRYPT 2005*. 2005, pp. 19–35.

[77] Andrew Chi-Chih Yao. "Protocols for Secure Computations (Extended Abstract)." In: *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. 1982, pp. 160–164.

[78] Yuliang Zheng, Tsutomu Matsumoto, and Hideki Imai. "On the Construction of Block Ciphers Provably Secure and Not Relying on Any Unproved Hypotheses." In: *Advances in Cryptology - CRYPTO '89*. 1989, pp. 461–480.

Part II

PUBLICATIONS

Practical Attacks on AES-like Cryptographic Hash Functions

Publication Information

Stefan Kölbl and Christian Rechberger. "Practical Attacks on AES-like Cryptographic Hash Functions." In: *Progress in Cryptology - LATINCRYPT 2014*. 2014, pp. 259–273

Contribution

- Main author.

Remarks

This publication has been slightly edited to fit the format.

Practical Attacks on AES-like Cryptographic Hash Functions

Stefan Kölbl and Christian Rechberger

Technical University of Denmark
stek@dtu.dk

Abstract. Despite the great interest in rebound attacks on AES-like hash functions since 2009, we report on a rather generic, albeit keyschedule dependent, algorithmic improvement: A new message modification technique to extend the inbound phase, which even for large internal states makes it possible to drastically reduce the complexity of attacks to very practical values for reduced-round versions. Furthermore, we describe new and practical attacks on Whirlpool and the recently proposed GOST R hash function with one or more of the following properties: more rounds, less time/memory complexity, and more relevant model. To allow for easy verification, we also provide a source-code for them.

Keywords: hash functions, cryptanalysis, collisions, Whirlpool, GOST R, Streebog, practical attacks

1 Introduction

Cryptographic hash functions are one of the most versatile primitives and have many practical applications like integrity checks, message authentication, digital signature or password protection. Often they are a critical part of more complex systems whose security might fall apart if hash a function does not provide the properties we expect it to have.

Cryptographic hash functions take as input a string of arbitrary finite length and produce a fixed-sized output of n bits called hash. As a consequence, the following main security requirements are defined for cryptographic hash functions:

- **Preimage Resistance:** For a given output y it should be computationally infeasible to find any input x' such that $y = h(x')$.

- **Second Preimage Resistance:** For given $x, y = h(x)$ it should be computationally infeasible to find any $x' \neq x$ such that $y = h(x')$.
- **Collision Resistance:** It should be computationally infeasible to find two distinct inputs x, x' such that $h(x) = h(x')$.

For any ideal hash function with n-bit output size, we can find preimages or second preimages with a complexity of 2^n, and collisions with a complexity of $2^{n/2}$ using generic attacks.

Most cryptographic hash functions are constructed iteratively by splitting the message into evenly sized blocks m_i and using a compression function f to update the state. We call the intermediate results x_i chaining values and the final output h hash value.

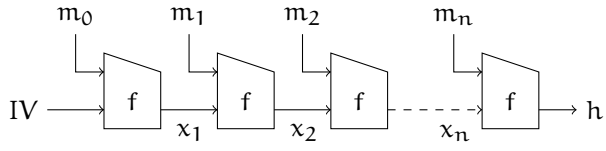


Figure 1: Iterative construction for a cryptographic hash function.

The security proofs for the hash function rely on the difficulty of finding a collision for this compression function, hence it is also of interest to consider the properties of the compression function and find properties which distinguish it from an ideal function.

- **semi-free start collision:** Find x, m, m' such that $f(x, m) = f(x, m')$.
- **free-start collision:** Find x, x', m, m' such that $f(x, m) = f(x', m')$.
- **near collision:** Find x, m, m' such that $f(x, m) \oplus f(x, m')$ has a low Hamming weight.

To sum up the various types with respect to their relevance: a semi-free-start collision is more interesting than a free-start collision, and a collision is more interesting than a near-collision.

1.1 Motivation

Cryptanalytic attacks are often hard to verify. Cryptanalysts often concentrate on the total running time of the attack, which is boiled down to a single number. While one can argue about the exact transition point between cryptanalytic attacks of practical and theoretical time complexity, it is often placed

around an equivalent of 2^{64} calls to the primitive [5]. While this is a reasonable assumption for state-level adversaries, it is out of reach for academic research labs. However, the ability to fully implement and verify attacks is crucial, as this is often the only way to make sure that all details are modelled correctly in the theoretical analysis. In this paper we therefore aim at attacks that can actually be executed (and verified) with limited budget computing resources.

In this paper we show a new practical attack on a class of AES-like hash functions. We show attacks on reduced round versions of the ISO/IEC 10118-3 standard `Whirlpool` [3] and `GOST R 34.11-2012` which is the new Russian federal standard [7]. The model we consider is semi-free-start attacks on the compression function, which in contrast to the free-start attacks do not allow the attacker to choose different chaining values in a pair of inputs. This reduced degree of freedom makes the task of cryptanalysts harder, but is more relevant as it is closer to the actual use in the hash function.

1.2 Contribution

Despite a lot of attention on rebound-attacks of AES and AES-like primitives, we show that more improvements are possible in the inbound phase.

To the best of our knowledge, currently no practical attacks on reduced round `GOST R` have been published. However, there exists a practical 4-round free-start collision attack on the `Whirlpool` compression function [20]. It seems very hard to apply this specific attack directly to `GOST R` due to the extra round in the key schedule, which gives `GOST R` additional security against these free-start attacks.

In this paper we show a new method to carry out a 4-round practical attack on the `Whirlpool` and `GOST R` compression function. Additionally, and in contrast to many other attacks known on `GOST R`, we do not need the freedom to add half a round at the end to turn a near-collision into a collision. As the full hash function also does not end with a half round, we argue that a result on 4 rounds can actually be more informative than a result on 4.5 rounds.

New message modification technique. The attack is based on the rebound attack and start-in-the-middle techniques, and it carefully chooses the key input to significantly reduce the complexity resulting in a very low complexity[1]. We are also able to improve the results on 6.5 rounds by extending this attack. We give an actual example for such a collision, and have the source code of both the attack and the general framework publicly available to facilitate further research on practical attacks[2]. The method is not specific to a

[1] Naturally, the improvement is not applicable for constructions or modes that do not allow modification of the key input

[2] The source-code can be found at `https://github.com/kste/aeshash`

particular primitive, but is an algorithmic technique that however depends on two conditions in a primitive to hold (see also Section 4).

1.3 Related Work

In Table 1 we summarize the practical results on `Whirlpool` and `GOST R`. As the `GOST R` compression function uses a design similar to the `Whirlpool` hash function [3], many of the previous results on `Whirlpool` can be applied to `GOST R`. We would also like to note on adding half a round at the end for `GOST R`. This does not always make an attack more difficult, and in some cases it makes it easier, as it makes it possible to turn a near-collision into a collision, therefore we distinguish for our attacks if it applies for both cases.

Table 1: Summary of attacks with a complexity up to 2^{64} on AES-based hash functions. Time is given in compression function calls and memory in bytes.

Function	Rounds	Time	Memory	Type	Reference
`GOST R`	4.5	2^{64}	2^{16}	semi-free-start collision	[21]
	4.75	practical	2^{8}	semi-free-start near-collision	[2]
	4	$2^{19.8}$	2^{16}	semi-free-start collision	this work
	4.5	$2^{19.8}$	2^{16}	semi-free-start collision	this work
	5.5	2^{64}	2^{64}	semi-free-start collision	[21]
	6.5	2^{64}	2^{16}	semi-free-start collision	this work
`Whirlpool`	4	$2^{25.1}$	2^{16}	semi-free-start collision	this work
	6.5	$2^{25.1}$	2^{16}	semi-free-start near-collision	this work
	4	2^{8}	2^{8}	free-start collision	[21]
	7	2^{64}	2^{8}	free-start collision	[20]

There have also been practical attacks on other AES-based hash functions like `Maelstroem` (6 out of 10 rounds [12]), `Grøstl` (6 out of 10 rounds [17]) and `Whirlwind` (4.5 out of 12 rounds [4]).

1.4 Rebound Attacks

The rebound attack is a powerful tool in the cryptanalysis of hash functions, especially for finding collisions for AES-based designs [14, 18]. The cipher is split into three sub-ciphers

$$E = E_{fw} \circ E_{in} \circ E_{bw}$$

and the attack proceeds in two steps. First, the inbound phase which is an efficient meet-in-middle in E_{in} using the available degree of freedom. This is followed by a probabilistic part, the outbound phase in E_{fw} and E_{bw} using the solutions from the inbound phase. The basic 4-round rebound attack uses

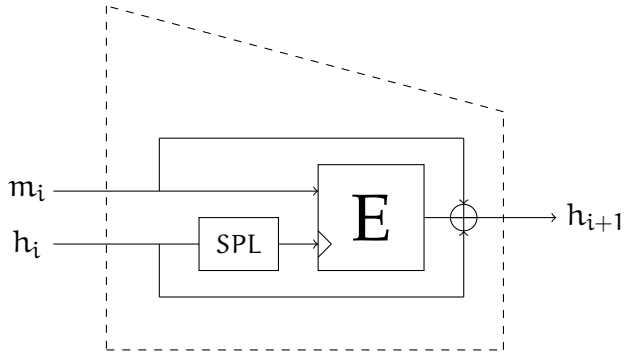


Figure 1: An outline of the GOST R compression function. The chaining input is processed through an additional round before entering E

a differential characteristic with $1-8-64-8-1$ active bytes per round and has a complexity of 2^{64}. There are many techniques extending and improving this attack. Some can even improve this very basic and simple setting of a square geometry, like start-from-the-middle [17], super S-box [8, 13] or solving three fully active states in the middle [9, 10]. Other generic extensions exploit additional degrees of freedom or non-square geometries to improve results, like and using multiple inbounds [13, 16]. In these settings, improved list-matching techniques [6, 19] are also a generic improvement.

2 Description of GOST R

This section gives a short description of the GOST R compression function as we will use it for describing our attack in detail. As we are only looking at the compression function, we leave out some details not relevant for the upcoming attack in order to simplify the description. For a more detailed description of GOST R we refer to [7].

The compression function g uses two 512-bit inputs (the message block m and the chaining value h) to update the state in the following way (see Figure 1)

$$g_N(h, m) = E(L \circ P \circ S(h), m) \oplus h \oplus m \tag{1}$$

where E is an AES-based block cipher using a state of 8×8 bytes and S, P, L are the same functions as used in this block cipher (see below).

If we want to find a collision for the compression function, the following equation must hold

$$\Delta m_i \oplus \Delta h_i \oplus \Delta E(h_i, m_i) = 0 \tag{2}$$

2.1 Block Cipher E

The block cipher E takes two 512-bit inputs M and K^0 and produces a 512-bit output C. The state update consists of 12 rounds r and a final key addition.

$$\begin{aligned} L_1 &= L \circ P \circ S \circ AK(M, K^0) \\ L_{i+1} &= L \circ P \circ S \circ AK(L^i, K^i) \quad i = 1 \ldots 11 \\ C &= AK(L^{12}, K^{12}) \end{aligned}$$

The following four operations are used in one round (see Figure 2):

- **AK** Adds the key byte-wise by XORing it to the state.
- **S** Substitutes each byte of the state independently using an 8-bit S-box.
- **P** Transposes the state.
- **L** Multiplies each row by an 8×8 MDS matrix.

The 512-bit key input is expanded to 13 subkeys $K_0, \ldots, K_{12}$. This is done similar to the state update but AK is replaced with the addition of a round-dependent constant RC^r.

$$\begin{aligned} L_{i+1} &= L \circ P \circ S \circ AK(K^0, RC^0) \quad i = 0 \ldots 11 \\ K^{12} &= AK(L^{12}, K^{12}) \end{aligned}$$

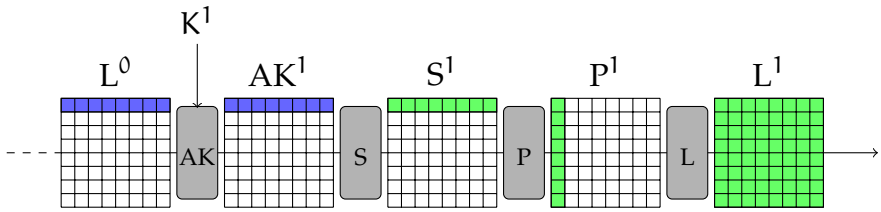


Figure 2: The four operations used in one round of `GOST R`.

2.2 Notation

The notation we use for naming the states is:

- The state after applying the round function $\{AK, S, P, L\}$ in round r is named $\{AK^r, S^r, P^r, L^r\}$

- The byte at position x, y in state X^r is named $X^r_{x,y}$
- A row is denoted by $X^r_{*,y}$ and a column by $X^r_{x,*}$
- ■ and ■ denote that there is a difference in a byte.
- ■ and ■ are used for highlighting values of a byte.

2.3 Differential Properties

The attacks in this paper are based on differential cryptanalysis, and the resulting complexity correlates with the differential properties of the round functions. Therefore, to ease understanding, we give a short overview of the properties that are relevant for our attack.

The linear layer L has a strong influence on the number of active S-boxes. There is no proof given that the linear layer L is MDS or has a branch number of 9 in the `GOST R` reference [7], but it was shown that this is the case in [11]. Hence, if we have one active byte at the input we will get 8 active bytes at the output with probability one. If we have a active bytes at the input the probability that there will be b active bytes at the output under the condition $a \neq 0, b \neq 0$ and $a + b \geqslant 9$ is $2^{(b-8)8}$.

The properties of the S-box have a strong influence on the complexity of our attack, as will be seen later. Given a S-box $S : \mathbb{F}_2^n \to \mathbb{F}_2^n$

$$\{x \mid S(x) \oplus S(x \oplus a) = b\} \tag{3}$$

is the number of solutions for an input a and output difference b. Table 2 gives the number of solutions for some S-box designs used in practice.

To get a bound on the probability of the differential characteristics we are interested in the maximum value of Equation 3 which we will refer to as the *maximum differential probability* (mdp) of an S-box. A 4-round differential characteristic has at least 81 active bytes due to the properties of the linear layer, therefore any 4-round characteristic has a probability of $\leqslant \text{mdp}^{81}$.

For the rebound attack it is also important to know the average number of possible output differences, given a non-zero input difference. We will refer to this as the average number of solutions (ANS) for an S-box which can be computed by constructing the *differential distribution table* (DDT). The ANS corresponds to the average number of non-zero entries in each row of the DDT.

This property influences the complexity for the matching step in the inbound phase and increases the costs of finding one solution. For the `GOST R` S-box we get on average 107.05 solutions.

Table 2: Comparison of different 8-bit S-box designs used in AES-based hash functions.

Solutions	AES	Whirlpool	GOST R
0	33150	39655	38235
2	32130	20018	22454
4	255	5043	4377
6	-	740	444
8	-	79	25
256	1	1	1

3 Attack on GOST R

In this section we describe our 4-round practical attack in detail and also show how it can be applied to more rounds. The description of the attack is split into two parts. First, we find a differential characteristic leading to a collision. Then we show how to construct a message pair following this characteristic in a very efficient way.

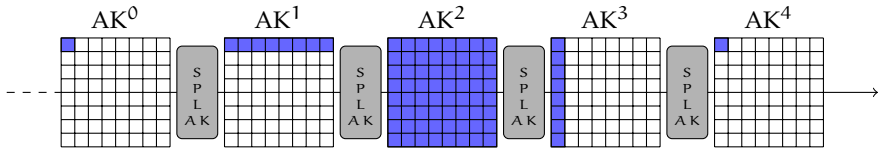


Figure 1: The 4-round differential characteristic used in our attack.

3.1 Constructing the Differential Characteristic

For our 4-round attack we use a characteristic of the form $1-8-64-8-1$ (see Figure 1). This truncated differential path has the minimal number of possible active S-boxes for 4 rounds and is the starting point for many attacks. Next, we will determine the values of the differences before continuing with the construction of the message pair.

The approach we use for this is based on techniques from the rebound attack, like the start-in-the-middle technique used in [17]. This approach would also give us an efficient way to find both the characteristic and message pair for a characteristic of the form $1-8-64-8$. However this would still lead to a higher attack complexity if extended to 4 rounds. Hence, we only use ideas from this approach to determine the differential characteristic and do not assume the key input as constant.

Precomputation.

First we pre-compute the differential distribution table (DDT) of the S-box and we also construct a list M_{lin}. This list contains all possible 255 non-zero values of $P_{0,0}$ and the result after applying L (see Figure 2).

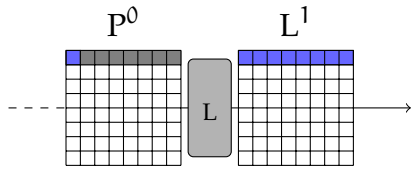


Figure 2: Computing list M_{lin} for all 255 values of $P^0_{0,0}$ (blue) to find all possible transitions from 1 to 8 bytes. Gray bytes are set to zero.

Construction.

1. Start with a random difference in $AK^4_{0,0}$ and propagate it back to S^2 through the inverse round functions. For the linear steps this is deterministic, and for propagating through the S-box we choose a random possible input difference to the given output difference. After this step we will have a full active state in S^2.

2. For each difference in S^2 we look up the set of all possible input differences from the DDT for each byte of the state.

3. Check for each row of AK^2 whether there is a possible match with the rows stored in M_{lin} (see Figure 3).

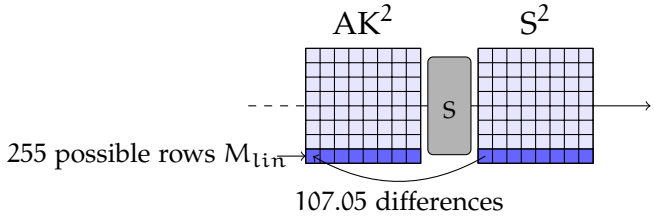


Figure 3: The matching step in the middle is done on each row individually. There are 2^8 possible values for each row $AK^2_{*,j}$ for $j = 0, 1, \ldots, 7$.

- The probability that a single byte matches is $107.05/255 \approx 2^{-1.252}$ therefore a row matches with a probability of $2^{-10.018}$.

- If we take into account that M_{lin} has 255 entries we expect to find a match with a probability of $1-(1-2^{-10.018})^{255} \approx 2^{-2.2}$.
- Therefore the probability for a match of all 8 rows is given by

$$(2^{-2.2})^8 = 2^{-17.6} \tag{1}$$

After this step we have found a characteristic spanning from S^1 to AK^4. Now we have to repeat the previous process for a single row to find the right differences in AK^1. This has a probability of $2^{-2.2}$ of succeeding. Hence we need to repeat the whole process $2^{19.8}$ times to obtain one solution.

Note that we can only choose 255 differences for $AK^4_{0,0}$, but we can also freely choose from the set of possible differences when propagating from S^3 to AK^3. This gives us an additional 107.05 choices for each row in S^2 leading to $\approx 2^{54}$ possible values for the state S^2. Hence, we have enough starting points for finding our differential characteristic.

3.2 Finding the Message Pair

Now we want to find a message pair which follows the previously constructed characteristic. At this point only the differences, but not the values of the state, are fixed. We start by fixing the values of AK^2 such that the 64 differential transitions $S^2 = S(AK^2)$ are fulfilled.

Next we use the key input to solve any further conditions on the active S-boxes in order to lower the complexity of our attack. This step is split into solving the conditions on $S^1_{*,0} = S(AK^1_{*,0})$ and $S^3_{0,*} = S(AK^3_{0,*})$.

Solving Conditions at the Start.

We have 8 conditions on $S^1_{*,0}$ which we need to solve. These conditions can be solved row-wise by choosing the corresponding values in K^2 such that $P^{-1}(L^{-1}(AK^2 \oplus K^2)) = S^1$. We can do this step row-wise by solving a linear equation. As there is only a single byte condition for each row, we only need one byte in the corresponding row of K^2 to solve the equation (see Figure 4). The remaining bytes are fixed to arbitrary values as we have no further conditions to fulfill at this step. These bytes could be used to solve more conditions for other differential characteristics or to construct additional solutions, as we will do for extending the attack on more rounds.

In this step we can generate up to 2^{56} solutions per row. Note that we only do this step for 7 rows, as we need the last row in the next step.

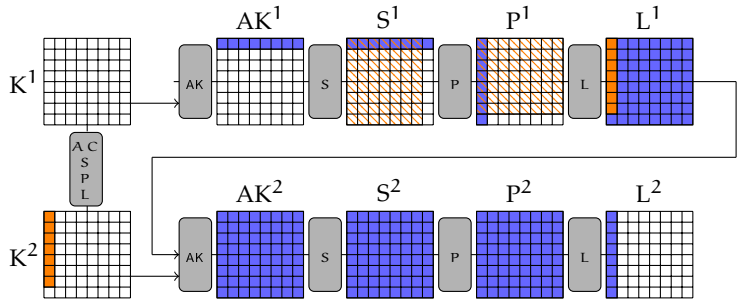


Figure 4: The values of AK^2 are fixed. We solve 7 of the conditions on S^1 by using the freedom in K^2 (bytes marked orange), which allows us to influence the values on the bytes in S^1 (orange slash pattern).

Solving Conditions at the End.

For solving the conditions $S^3 = S(AK^3)$, we can use the bytes in $K^2_{*,7}$. These bytes form a column in $\mathrm{cmssKP}^3_{7,*}$ (see Figure 5), which allows us to solve a single byte condition per row for AK^3.

1. Assume that $K^2_{*,0-6}$ are fixed and propagate them forward to KP^3.
2. We can now solve the conditions for each row individually. In each row there are 7 bytes fixed in KP^3 and a single byte in K^3 (from AK^3). This gives us a linear equation with one solution per row and allows us to solve all conditions on AK^3.

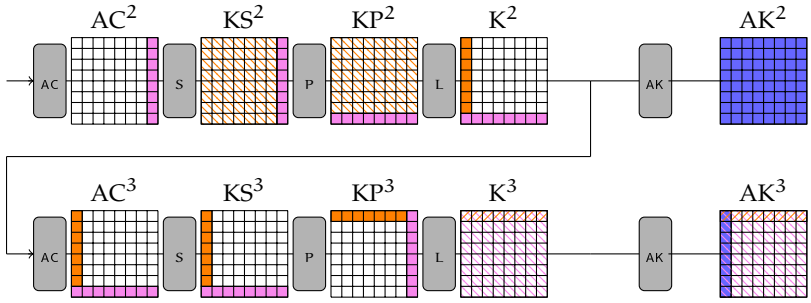


Figure 5: Solving all the conditions on AK^3. The orange values are fixed from the previous step and the purple values are used to fulfill the conditions on AK^3.

Remaining Conditions.

We still need to solve one byte condition on $S^1_{0,7}$, which can be done by repeating the previous procedure 2^8 times. The bytes which are used to solve the conditions on AK^3 form a row in K^2 and influence the values of L^1 resp. P^1 and S^1 (see Figure 1 in Section A). This implies that we can change the value of $S^1_{0,7}$ by constructing different solutions for $K^2_{*,7}$.

The only remaining condition is $\Delta AK^0_{0,0} = \Delta AK^4_{0,0}$, which can again be solved by repeating the previous steps 2^8 times. It follows that we need to repeat the algorithm shown in Section 3.2 about 2^{16} times.

Complexity.

We can construct the differential characteristic with a complexity of $2^{19.8}$. Finding a message pair following this characteristic requires 2^{16} steps using our message modification technique. Hence, the total complexity of the attack is $\approx 2^{19.9}$. We have implemented this attack and verified our results. The un-optimized proof-of-concept implementation in Python is publicly available [1]. An example for a 4-round collision can be found in Section B.

3.3 Extending the Attack

As we only need to control 15 bytes of the key, we can extend the attack on 6.5 rounds by using a characteristic of the form $8-1-8-64-8-1-8$. In this case we would use the same approach to find the differential characteristic for 4 rounds and in the message modification part we would construct more solutions by using the additional freedom in the key. This will influence the differences at the input/output of the 6.5 rounds. The complexity of this attack is $\approx 2^{64}$, as the 8-byte difference at the input/output needs to be equal.

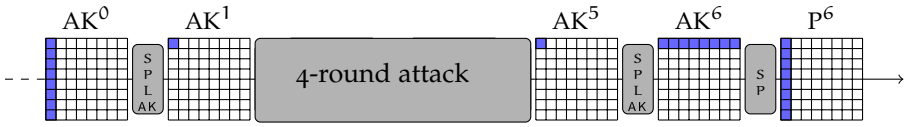


Figure 6: The 4-round attack is extended by one round in the beginning and one round in the end to mount an attack on 6.5 rounds.

4 Application to other AES-based Hash Functions

The message modification technique presented is not specific to GOST R, but requires a few criteria to be met. First the transposition layer has to have the property that every byte of a single row/column is moved to a different row/column (see Figure 1). This is true for all AES-based hash functions we consider in this paper, as it is a desired property against other attacks.

The second criteria is that there is a key addition in every round, hence our attack is applicable to both Whirlpool and GOST R. Permutation-based designs like Grøstl do not have this property. The attacker has less control of the input for each round, which makes the hash function more resistant against these types of attacks.

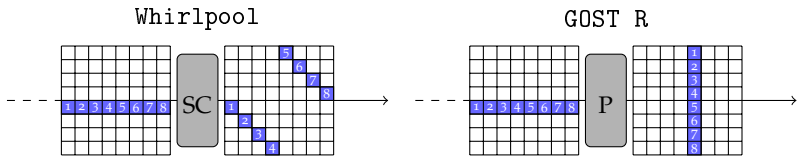


Figure 1: The transposition layer used in Whirlpool and GOST R.

The complexity of the attack depends on the choice of the S-box, as this directly influences the costs of constructing the differential characteristic. Given the average number of solutions $\bar{s}$ for $\Delta out = S(\Delta in)$ with a fixed value Δin, this directly gives the complexity for the matching step of the attack

$$\left(1 - \left(1 - \frac{\bar{s}}{255}\right)^{255}\right)^{8} \tag{1}$$

and the number of possible states for S^2 is $\approx \bar{s}^{\,8}$. A comparison of the different S-boxes used in AES-based hash functions is given in Table 3.

Table 3: Comparing the maximum differential probability (MDP) and average number of solutions (ANS) for different 8-bit S-boxes in AES-based designs.

S-box	MDP	ANS	Matching Costs	$\#S^2$
AES	2^{-6}	127	$2^{6.42}$	$2^{55.91}$
Whirlpool	2^{-5}	101.49	$2^{25.10}$	$2^{53.32}$
GOST-R	2^{-5}	107.05	$2^{19.77}$	$2^{53.94}$

5 Conclusion

In this paper, we have shown new practical attacks for both the `Whirlpool` and `GOST R` compression function. We presented a 4-round attack with very low complexity of $2^{25.10}$ resp. $2^{19.8}$. Importantly, the attack is fully verified and source-code for it is available. In the case of `GOST R` the attack can be extended to find collisions for 6.5 rounds with a complexity of 2^{64} and for `Whirlpool` we can extend it to construct a near-collision in 50 bytes with a complexity of $2^{25.10}$ for 6.5 rounds of the compression function. The difference in the results for `GOST R` and Whirlpool is due to the ShiftColumns operation which does not align the bytes to lead to a collision for the differential characteristic we use.

Our attack is applicable to all AES-based primitives where it is possible for the attacker to control the key input for a few rounds. This significantly reduces the complexity of previous attacks and might be useful to speed up other attacks on AES-based hash-function designs.

References

[1] https://github.com/kste/aeshash.

[2] Riham AlTawy, Aleksandar Kircanski, and Amr M. Youssef. *Rebound Attacks on Stribog*. Cryptology ePrint Archive, Report 2013/539. http://eprint.iacr.org/. 2013.

[3] Paulo Barreto and Vincent Rijmen. "The Whirlpool hashing function." In: *First open NESSIE Workshop, Leuven, Belgium*. Vol. 13. 2000, p. 14.

[4] Paulo S. L. M. Barreto, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Elmar Tischhauser. "Whirlwind: a new cryptoaphic hash function." In: *Des. Codes Cryptography* 56.2-3 (2010), pp. 141–162.

[5] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. "Key Recovery Attacks of Practical Complexity on AES-256 Variants with up to 10 Rounds." In: *EUROCRYPT*. Ed. by Henri Gilbert. Vol. 6110. Lecture Notes in Computer Science. Springer, 2010, pp. 299–319. ISBN: 978-3-642-13189-9.

[6] Itai Dinur, Orr Dunkelman, Nathan Keller, and Adi Shamir. "Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems." In: *CRYPTO*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 719–740. ISBN: 978-3-642-32008-8.

[7] V. Dolmatov and A. Degtyarev. *GOST R 34.11-2012: Hash Function*. http://tools.ietf.org/html/rfc6986. 2013.

[8] Henri Gilbert and Thomas Peyrin. "Super-Sbox Cryptanalysis: Improved Attacks for AES-Like Permutations." In: *FSE*. Ed. by Seokhie Hong and Tetsu Iwata. Vol. 6147. Lecture Notes in Computer Science. Springer, 2010, pp. 365–383. ISBN: 978-3-642-13857-7.

[9] Jérémy Jean, María Naya-Plasencia, and Thomas Peyrin. "Improved Rebound Attack on the Finalist Grøstl." In: *FSE*. Ed. by Anne Canteaut. Vol. 7549. Lecture Notes in Computer Science. Springer, 2012, pp. 110–126. ISBN: 978-3-642-34046-8.

[10] Jérémy Jean, María Naya-Plasencia, and Thomas Peyrin. "Improved Cryptanalysis of AES-like Permutations." In: *J. Cryptology* 27.4 (2014), pp. 772–798. DOI: 10.1007/s00145-013-9156-7. URL: http://dx.doi.org/10.1007/s00145-013-9156-7.

[11] Oleksandr Kazymyrov and Valentyna Kazymyrova. *Algebraic Aspects of the Russian Hash Standard GOST R 34.11-2012*. Cryptology ePrint Archive, Report 2013/556. http://eprint.iacr.org/. 2013.

[12] Stefan Kölbl and Florian Mendel. "Practical Attacks on the Maelstrom-0 Compression Function." In: *ACNS*. Ed. by Javier Lopez and Gene Tsudik. Vol. 6715. Lecture Notes in Computer Science. 2011, pp. 449–461. ISBN: 978-3-642-21553-7.

[13] Mario Lamberger, Florian Mendel, Christian Rechberger, Vincent Rijmen, and Martin Schläffer. "Rebound Distinguishers: Results on the Full Whirlpool Compression Function." In: *ASIACRYPT*. Ed. by Mitsuru Matsui. Vol. 5912. Lecture Notes in Computer Science. Springer, 2009, pp. 126–143. ISBN: 978-3-642-10365-0.

[14] Mario Lamberger, Florian Mendel, Martin Schläffer, Christian Rechberger, and Vincent Rijmen. "The Rebound Attack and Subspace Distinguishers: Application to Whirlpool." English. In: *Journal of Cryptology* (2013), pp. 1–40. ISSN: 0933-2790. DOI: 10.1007/s00145-013-9166-5. URL: http://dx.doi.org/10.1007/s00145-013-9166-5.

[15] Mitsuru Matsui, ed. *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*. Vol. 5912. Lecture Notes in Computer Science. Springer, 2009. ISBN: 978-3-642-10365-0.

[16] Krystian Matusiewicz, María Naya-Plasencia, Ivica Nikolic, Yu Sasaki, and Martin Schläffer. "Rebound Attack on the Full Lane Compression Function." In: *ASIACRYPT*. Ed. by Mitsuru Matsui. Vol. 5912. Lecture Notes in Computer Science. Springer, 2009, pp. 106–125. ISBN: 978-3-642-10365-0.

[17] Florian Mendel, Thomas Peyrin, Christian Rechberger, and Martin Schläffer. "Improved Cryptanalysis of the Reduced Grøstl Compression Function, ECHO Permutation and AES Block Cipher." In: *Selected Areas in Cryptography*. Ed. by Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini. Vol. 5867. Lecture Notes in Computer Science. Springer, 2009, pp. 16–35. ISBN: 978-3-642-05443-3.

[18] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. "The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl." In: *FSE*. Ed. by Orr Dunkelman. Vol. 5665. Lecture Notes in Computer Science. Springer, 2009, pp. 260–276. ISBN: 978-3-642-03316-2.

[19] María Naya-Plasencia. "How to Improve Rebound Attacks." In: *CRYPTO*. Ed. by Phillip Rogaway. Vol. 6841. Lecture Notes in Computer Science. Springer, 2011, pp. 188–205. ISBN: 978-3-642-22791-2.

[20] Yu Sasaki, Lei Wang, Shuang Wu, and Wenling Wu. "Investigating Fundamental Security Requirements on Whirlpool: Improved Preimage and Collision Attacks." In: ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Springer Berlin Heidelberg, 2012, pp. 562–579.

[21] Zongyue Wang, Hongbo Yu, and Xiaoyun Wang. *Cryptanalysis of GOST R Hash Function*. Cryptology ePrint Archive, Report 2013/584. http://eprint.iacr.org/. 2013.

A Solving Conditions

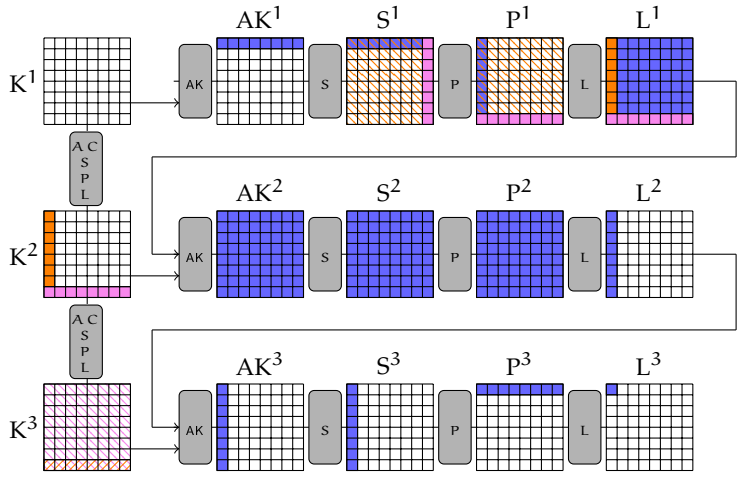


Figure 1: Solving both conditions on S^1 and AK^3. The bytes marked purple solve the conditions on AK^3 and a single condition on S^1, whereas the orange bytes solve 7 conditions on S^1.

B Colliding Message Pair

Here a colliding message pair (M, M') and the chaining value are given. The message pair has been found by using the 4-round characteristic and the difference in the messages is $\Delta AK^0_{0,0} = \Delta AK^4_{0,0} =$ fc. All values are given in hexadecimal notation.

K^0

```
e80c313d6875c049
865604acdb024055
5cfd18c5d1a5a3f3
793359475ae90836
528e80e4aedfd674
20c777c0306ff6da
da4f9acc1e4fc9c4
04f486b91fa6bdeb
```

AK^0	AK'^0	ΔAK^0
81c2cf897a89e94f	7dc2cf897a89e94f	fc00000000000000
11e6c3ac6020a3c9	11e6c3ac6020a3c9	0000000000000000
ef44c4305cfade20	ef44c4305cfade20	0000000000000000
c04dc08f87db9f8f	c04dc08f87db9f8f	0000000000000000
b56983ec66229993	b56983ec66229993	0000000000000000
1bf0262cb3b92956	1bf0262cb3b92956	0000000000000000
dcb1e1511414ac83	dcb1e1511414ac83	0000000000000000
8efcd0ef76fa4671	8efcd0ef76fa4671	0000000000000000

AK^4	AK'^4	ΔAK^4
1444aab6eb146b46	e844aab6eb146b46	fc00000000000000
0692f61cf6b43022	0692f61cf6b43022	0000000000000000
935d2a8720ae7f0f	935d2a8720ae7f0f	0000000000000000
329542c23a812da1	329542c23a812da1	0000000000000000
eca3ba2984ab0219	eca3ba2984ab0219	0000000000000000
636b834737911049	636b834737911049	0000000000000000
0ce0d7edefe72c55	0ce0d7edefe72c55	0000000000000000
6dc8a467a81f8587	6dc8a467a81f8587	0000000000000000

Observations on the SIMON Block Cipher Family

Publication Information

Stefan Kölbl, Gregor Leander, and Tyge Tiessen. "Observations on the SIMON Block Cipher Family." In: *Advances in Cryptology - CRYPTO 2015*. 2015, pp. 161–185

Contribution

- Main contributions are Section 5, 6.

Remarks

This publication has been slightly edited to fit the format.

Observations on the SIMON block cipher family

Stefan Kölbl[1], Gregor Leander[2], and Tyge Tiessen[1]
stek@dtu.dk, gregor.leander@rub.de, tyti@dtu.dk

[1] DTU Compute, Technical University of Denmark, Denmark
[2] Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany

Abstract. In this paper we analyse the general class of functions underlying the SIMON block cipher. In particular, we derive efficiently computable and easily implementable expressions for the exact differential and linear behaviour of SIMON-like round functions.

Following up on this, we use those expressions for a computer aided approach based on SAT/SMT solvers to find both optimal differential and linear characteristics for SIMON. Furthermore, we are able to find all characteristics contributing to the probability of a differential for SIMON32 and give better estimates for the probability for other variants.

Finally, we investigate a large set of SIMON variants using different rotation constants with respect to their resistance against differential and linear cryptanalysis. Interestingly, the default parameters seem to be not always optimal.

Keywords: SIMON, differential cryptanalysis, linear cryptanalysis, block cipher, Boolean functions

1 Introduction

Lightweight cryptography studies the deployment of cryptographic primitives in resource-constrained environments. This research direction is driven by a demand for cost-effective, small-scale communicating devices such as RFID tags that are a cornerstone in the Internet of Things. Most often the constrained resource is taken to be the chip-area but other performance metrics such as latency [7], code-size [2] and ease of side-channel protection [12]) have been considered as well. Some of these criteria were already treated in NOEKEON [9].

The increased importance of lightweight cryptography and its applications has lately been reflected in the NSA publishing two dedicated lightweight

cipher families: Simon and Speck [5]. Considering that this is only the third time within four decades that the NSA has published a block cipher, this is quite remarkable. Especially as NIST has started shortly after this publication to investigate the possibilities to standardise lightweight primitives, Simon and Speck certainly deserve a thorough investigation. This is emphasised by the fact that, in contrast to common practice, neither a security analysis nor a justification of the design choices were published by the NSA. This lack of openness necessarily gives rise to curiosity and caution.

In this paper we focus on the Simon family of block ciphers; an elegant, innovative and very efficient set of block ciphers. There exists already a large variety of papers, mainly focussed on evaluating Simon's security with regard to linear and differential cryptanalysis. Most of the methods used therein are rather ad-hoc, often only using approximative values for the differential round probability and in particular for the linear square correlation of one round.

Our Contribution

With this study, we complement the existing work threefold. Firstly we develop an exact closed form expression for the differential probability and a $\log(n)$ algorithm for determining the square correlation over one round. Their accuracy is proven rigorously. Secondly we use these expressions to implement a model of differential and linear characteristics for SAT/SMT solvers which allows us to find the provably best characteristics for different instantiations of Simon. Furthermore we are able to shed light on how differentials in Simon profit from the collapse of many differential characteristics. Thirdly by generalising the probability expressions and the SAT/SMT model, we are able to compare the quality of different parameter sets with respect to differential and linear cryptanalysis.

As a basis for our goal to understand both the security of Simon as well as the choice of its parameter set, we rigorously derive formulas for the differential probabilities and the linear square correlations of the Simon-like round function that can be evaluated in constant time and time linear in the word size respectively. More precisely, we study differential probabilities and linear correlations of functions of the form

$$S^a(x) \odot S^b(x) + S^c(x)$$

where $S^i(x)$ corresponds to a cyclic left shift of x and $\odot$ denotes the bitwise AND operation.

We achieve this goal by first simplifying this question by considering equivalent descriptions both of the round function as well as the whole cipher

(cf. Section 2.4). These simplifications, together with the theory of quadratic boolean functions, result in a clearer analysis of linear and differential properties (cf. Section 3 and Section 4). Importantly, the derived simple equations for computing the probabilities of the SIMON round function can be evaluated efficiently and, more importantly maybe, are conceptually very easy. This allows them to be easily used in computer-aided investigations of differential and linear properties over more rounds. It should be noted here that the expression for linear approximations is more complex than the expression for the differential case. However, with respect to the running time of the computer-aided investigations this difference is negligible.

We used this to implement a framework based on SAT/SMT solvers to find the provably best differential and linear characteristics for various instantiations of SIMON (cf. Section 5, in particular Table 1). Furthermore we are able to shed light on how differentials in SIMON profit from the collapse of many differential characteristics by giving exact distributions of the probabilities of these characteristics for chosen differentials. The framework is open source and publicly available to encourage further research [13].

In Section 6 we apply the developed theory and tools to investigate the design space of SIMON-like functions. In particular, using the computer-aided approach, we find that the standard SIMON parameters are not optimal with regard to the best differential and linear characteristics.

As a side result, we improve the probabilities for the best known differentials for several variants and rounds of SIMON. While this might well lead to (slightly) improved attacks, those improved attacks are out of the scope of our work.

Interestingly, at least for SIMON32 our findings indicate that the choices made by the NSA are good but not optimal under our metrics, leaving room for further investigations and questions. To encourage further research, we propose several alternative parameter choices for SIMON32. Here, we are using the parameters that are optimal when restricting the criteria to linear, differential and dependency properties. We encourage further research on those alternative choices to shed more light on the undisclosed design criteria.

We also like to point out that the SIMON key-scheduling was not part of our investigations. Its influence on the security of SIMON is left as an important open question for further investigations. In line with this, whenever we investigate multi-round properties of SIMON in our work, we implicitly assume independent round keys in the computation of probabilities.

Finally, we note that most of our results can be applied to more general constructions, where the involved operations are restricted to AND, XOR, and rotations.

Related Work

There are various papers published on the cryptanalysis of SIMON [1, 3, 6, 17–19]. The most promising attacks so far are based on differential and linear cryptanalysis, however a clear methodology of how to derive the differential probabilities and square correlations seems to miss in most cases. Biryukov, Roy and Velichkov [6] derive a correct, but rather involved method to find the differential probabilities. Abed, List, Lucks and Wenzel [1] state an algorithm for the calculation of the differential probabilities but without further explanation. For the calculation of the square correlations an algorithm seems to be missing all together.

Previous work also identifies various properties like the strong differential effect and give estimate of the probability of differentials.

The concept behind our framework was previously also applied on the ARX cipher Salsa20 [14] and the CAESAR candidate NORX [4]. In addition to the applications proposed in previous work we extend it for linear cryptanalysis, examine the influence of rotation constants and use it to compute the distribution of characteristics corresponding to a differential.

2 Preliminaries

In this section, we start by defining our notation and giving a short description of the round function. We recall suitable notions of equivalence of Boolean functions that allow us to simplify our investigations of SIMON-like round functions. Most of this section is generally applicable to AND-RX constructions, i.e. constructions that only make use of the bitwise operations AND, XOR, and rotations.

2.1 Notation

We denote by $\mathbb{F}_2$ the field with two elements and by $\mathbb{F}_2^n$ the n-dimensional vector space over $\mathbb{F}_2$. By $\mathbf{0}$ and $\mathbf{1}$ we denote the vectors of $\mathbb{F}_2^n$ with all 0s and all 1s respectively. The Hamming weight of a vector $a \in \mathbb{F}_2^n$ is denoted as $\mathrm{wt}(a)$. By $\mathbb{Z}_n$ we denote the integers modulo n.

The addition in $\mathbb{F}_2^n$, i.e. bit-wise XOR, is denoted by $+$. By $\odot$ we denote the AND operation in $\mathbb{F}_2^n$, i.e. multiplication over $\mathbb{F}_2$ in each coordinate:

$$x \odot y = (x_i y_i)_i.$$

By $\vee$ we denote the bitwise OR operation. By $\overline{x}$ we denote the bitwise negation of x, i.e. $\overline{x} := (x + \mathbf{1})$. We denote by $S^i : \mathbb{F}_2^n \to \mathbb{F}_2^n$ the left circular shift

by i positions. We also note that any arithmetic of bit indices is always done modulo the word size n.

In this paper we are mainly concerned with functions of the form

$$f_{a,b,c}(x) = S^a(x) \odot S^b(x) + S^c(x) \tag{1}$$

and we identify such functions with its triple (a, b, c) of parameters.

For a vectorial Boolean function on n bits, $f : \mathbb{F}_2^n \to \mathbb{F}_2^n$, we denote by

$$\widehat{f}(\alpha, \beta) = \sum_x \mu\left(\langle \beta, f \rangle + \langle \alpha, x \rangle\right)$$

the Walsh (or Fourier) coefficient with input mask α and output mask β. Here we use $\mu(x) = (-1)^x$ to simplify the notation.

The corresponding squared correlation of f is given by

$$C^2(\alpha \to \beta) = \left(\frac{\widehat{f}(\alpha, \beta)}{2^n}\right)^2.$$

For differentials we similarly denote by $\Pr(\alpha \to \beta)$ the probability that a given input difference α results in a given output difference β, i.e.

$$\Pr(\alpha \to \beta) = \frac{|\{x \mid f(x) + f(x + \alpha) = \beta\}|}{2^n}.$$

Furthermore, $\mathsf{Dom}(f)$ is the domain of a function f, $\mathsf{Img}(f)$ is its image.

2.2 Description of SIMON

Simon is a family of lightweight block ciphers with block sizes 32, 48, 64, 96, and 128 bits. The constructions are Feistel ciphers using a word size n of 16, 24, 32, 48 or 64 bits, respectively. We will denote the variants as Simon2n. The key size varies between of 2, 3, and 4 n-bit words. The round function of Simon is composed of AND, rotation, and XOR operations on the complete word (see Figure 1). More precisely, the round function in Simon corresponds to

$$S^8(x) \odot S^1(x) + S^2(x),$$

that is to the parameters $(8, 1, 2)$ for f as given in Equation 1. As we are not only interested in the original Simon parameters, but in investigating the entire design space of Simon-like functions, we denote by

$$\textsc{Simon}[a, b, c]$$

the variant of Simon where the original round function is replaced by $f_{a,b,c}$ (cf. Equation 1).

As it is out of scope for our purpose, we refer to [5] for the description of the key-scheduling.

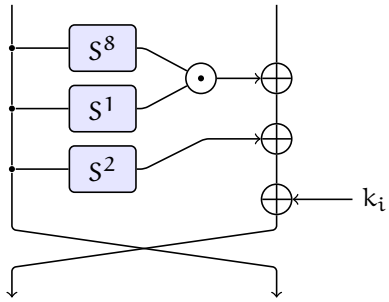


Figure 1: The round function of Simon.

2.3 Affine equivalence of Boolean Functions

Given two (vectorial) Boolean functions f_1 and f_2 on $\mathbb{F}_2^n$ related by

$$f_1(x) = (A \circ f_2 \circ B)(x) + C(x)$$

where A and B are affine permutations and C is an arbitrary affine mapping on $\mathbb{F}_2^n$ we say that f_1 and f_2 are *extended affine equivalent* (cf. [8] for a comprehensive survey).

With respect to differential cryptanalysis, if f_1 and f_2 are extended affine equivalent then the differential $\alpha \xrightarrow{f_1} \beta$ over f_1 has probability p if and only if the differential

$$B(\alpha) \xrightarrow{f_2} A^{-1}(\beta + C(\alpha))$$

over f_2 has probability p as well.

For linear cryptanalysis, a similar relation holds for the linear correlation. If f_1 and f_2 are related as defined above, it holds that

$$\widehat{f_1}(\alpha, \beta) = \widehat{f_2}\left(\left(C \circ B^{-1}\right)^T \beta + \left(B^{-1}\right)^T \alpha, A^T \beta\right).$$

Thus up to linear changes we can study f_2 instead of f_1 directly. Note that, for an actual attack, these changes are usually critical and can certainly not be ignored. However, tracing the changes is, again, simple linear algebra.

For differential and linear properties of Simon-like functions of the form

$$f_{a,b,c}(x) = S^a(x) \odot S^b(x) + S^c(x)$$

this implies that it is sufficient to find the differential and linear properties of the simplified variant

$$f(x) = x \odot S^d(x)$$

and then transfer the results back by simply using linear algebra.[3]

2.4 Structural Equivalence Classes in AND-RX Constructions

AND-RX constructions, i.e. constructions that make only use of the operations AND ($\odot$), XOR ($+$), and rotations (S^r), exhibit a high degree of symmetry. Not only are they invariant under rotation of all input words, output words and constants, they are furthermore structurally invariant under any affine transformation of the bit-indices. As a consequence of this, several equivalent representations of the SIMON variants exist.

Let T be a permutation of the bits of an n-bit word that corresponds to an affine transformation of the bit-indices. Thus there are $s \in \mathbb{Z}_n^*$ and $t \in \mathbb{Z}_n$ such that bit i is renamed to $s \cdot i + t$. As the AND and XOR operations are bitwise, T clearly commutes with these:

$$Tv \odot Tw = T(v \odot w)$$
$$Tv + Tw = T(v + w)$$

where v and w are n-bit words. A rotation to the left by r can be written bitwise as $S^r(v)_i = v_{i-r}$. For a rotation, we thus get the following bitwise relation after transformation with T

$$S^r(v)_{s\cdot i+t} = v_{s\cdot(i-r)+t} = v_{s\cdot i+t-s\cdot r} \ .$$

Substituting $s \cdot i + t$ with j this is the same as

$$S^r(v)_j = v_{j-s\cdot r} \ .$$

Thus the rotation by r has been changed to a rotation by $s \cdot r$. Thus we can write

$$TS^r v = S^{s\cdot r} Tv.$$

Commuting the linear transformation of the bit-indices with a rotation thus only changes the rotation constant by a factor. In the special case where all input words, output words and constants are rotated, which corresponds to the case $s = 1$, the rotation constant are left untouched.

To summarise the above, when applying such a transformation T to all input words, output words and constants in an AND-RX construction, the structure of the constructions remains untouched apart from a multiplication of the rotation constants by the factor s.

This means for example for SIMON32 that changing the rotation constants from $(8, 1, 2)$ to $(3 \cdot 8, 3 \cdot 1, 3 \cdot 2) = (8, 3, 6)$ and adapting the key schedule

[3] Note that we can transform the equation $f(x) = S^a(x) \odot S_b(x) + S^c(x)$ to the equation $S^{-a}(f(x)) + S^{c-a}(x)) = x \odot S^{b-a}(x)$.

accordingly gives us the same cipher apart from a bit permutation. As s has to be coprime to n, all s with $\gcd(s, n) = 1$ are allowed, giving $\varphi(n)$ equivalent tuples of rotation constants in each equivalence class where φ is Euler's phi function.

Together with the result from Section 2.3, this implies the following lemma.

Lemma 1. *Any function $f_{a,b,c}$ as defined in Equation 1 is extended affine equivalent to a function*

$$f(x) = x \odot S^d(x)$$

where $d|n$ or $d = 0$.

When looking at differential and square correlations of Simon-like round functions this means that it is sufficient to investigate this restricted set of functions. The results for these functions can then simply be transferred to the general case.

3 Differential Probabilities of SIMON-like round functions

In this section, we derive a closed expression for the differential probability for all Simon-like round functions, i.e. all functions as described in Equation 1. The main ingredients here are the derived equivalences and the observation that any such function is quadratic. Being quadratic immediately implies that its derivative is linear and thus the computation of differential probabilities basically boils down to linear algebra (cf. Theorem 2). However, to be able to efficiently study multiple-round properties and in particular differential characteristics, it is important to have a simple expression for the differential probabilities. Those expressions are given for $f(x) = x \odot S^1(x)$ in Theorem 3 and for the general case in Theorem 4.

3.1 A closed expression for the differential probability

The following statement summarises the differential properties of the f function.

Theorem 2. *Given an input difference α and an output difference β the probability p of the corresponding differential (characteristic) for the function $f(x) = x \odot S^a(x)$ is given by*

$$p_{\alpha,\beta} = \begin{cases} 2^{-(n-d)} & \text{if } \beta + \alpha \odot S^a(\alpha) \in \mathrm{Img}(L_\alpha) \\ 0 & \text{else} \end{cases}$$

where

$$d = \dim \ker(L_\alpha)$$

and

$$L_\alpha(x) = x \odot S^a(\alpha) + \alpha \odot S^a(x)$$

Proof. We have to count the number of solutions to the equation

$$f(x) + f(x + \alpha) = \beta.$$

This simplifies to

$$L_\alpha(x) = x \odot S^a(\alpha) + \alpha \odot S^a(x) = \beta + \alpha \odot S^a(\alpha)$$

As this is an affine equation, it either has zero solutions or the number of solutions equals the kernel size, i.e. the number of elements in the subspace

$$\{x \mid x \odot S^a(\alpha) + \alpha \odot S^a(x) = \mathbf{0}\}.$$

Clearly, the equation has solutions if and only if $\beta + \alpha \odot S^a(\alpha)$ is in the image of L_α. □

Next we present a closed formula to calculate the differential probability in the case where $a = 1$. Furthermore we restrict ourselves to the case where n is even.

Theorem 3. *Let*

$$\texttt{varibits} = S^1(\alpha) \vee \alpha$$

and

$$\texttt{doublebits} = \alpha \odot \overline{S^1(\alpha)} \odot S^2(\alpha).$$

Then the probability that difference α goes to difference β is

$$P(\alpha \to \beta) = \begin{cases} 2^{-n+1} & \text{if } \alpha = \mathbf{1} \text{ and } \mathrm{wt}(\beta) \equiv 0 \mod 2 \\ 2^{-\mathrm{wt}(\texttt{varibits}+\texttt{doublebits})} & \text{if } \alpha \neq \mathbf{1} \text{ and } \beta \odot \overline{\texttt{varibits}} = \mathbf{0} \\ & \text{and } (\beta + S^1(\beta)) \odot \texttt{doublebits} = \mathbf{0} \\ 0 & \text{else} \end{cases}$$

Proof. According to Theorem 2, we need to prove two things. Firstly we need to prove that the rank of L_α (i.e. $n - \dim \ker L_\alpha$) is $n - 1$ when $\alpha = \mathbf{1}$, and $\mathrm{wt}(\texttt{varibits} + \texttt{doublebits}))$ otherwise. Secondly we need to prove that $\beta + \alpha \odot S^1(\alpha) \in \mathrm{Img}(L_\alpha)$ iff $\mathrm{wt}(\beta) \equiv 0 \mod 2$ when $\alpha = \mathbf{1}$, and that $\beta + \alpha \odot S^1(\alpha) \in \mathrm{Img}(L_\alpha)$ iff $\beta \odot \texttt{varibits} = \mathbf{0}$ and $(\beta + S^1(\beta)) \odot \texttt{doublebits} = \mathbf{0}$ when $\alpha \neq \mathbf{1}$.

We first consider the first part. Let us write $L_\alpha(x)$ in matrix form and let us take x to be a column vector. $S^1(\alpha) \odot x$ can be written as $M_{S^1(\alpha)\odot} x$ with

$$M_{S^1(\alpha)\odot} = \begin{pmatrix} \alpha_{n-1} & \dots & \dots & 0 \\ \vdots & \alpha_0 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \dots & \dots & \alpha_{n-2} \end{pmatrix} . \tag{1}$$

Equivalently we can write $\alpha \odot x$ and $S^1(x)$ with matrices as $M_{\alpha\odot} x$ and $M_{S^1} x$ respectively where

$$M_{\alpha\odot} = \begin{pmatrix} \alpha_0 & \dots & \dots & 0 \\ \vdots & \alpha_1 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \dots & \dots & \alpha_{n-1} \end{pmatrix} \text{ and } M_{S^1} = \begin{pmatrix} 0_{1,n-1} & I_{1,1} \\ I_{n-1,n-1} & 0_{n-1,1} \end{pmatrix} , \tag{2}$$

i.e. M_{S^1} consists of two identity and two zero submatrices. The result of $M_{S^1(\alpha)\odot} + M_{\alpha\odot} M_{S^1}$ can now be written as

$$\begin{pmatrix} \alpha_{n-1} & 0 & 0 & \dots & \alpha_0 \\ \alpha_1 & \alpha_0 & 0 & \dots & 0 \\ 0 & \alpha_2 & \alpha_1 & & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \alpha_{n-1} & \alpha_{n-2} \end{pmatrix} \tag{3}$$

Clearly the rank of the matrix is $n-1$ when all α_i are 1. Suppose now that not all α_i are 1. In that case, a set of non-zero rows is linearly dependent iff there exist two identical rows in the set. Thus to calculate the rank of the matrix, we need to calculate the number of unique non-zero rows.

By associating the rows in the above matrix with the bits in `varibits`, we can clearly see that the number of non-zero rows in the matrices corresponds to the number of 1s in $\texttt{varibits} = S^1(\alpha) \vee \alpha$.

To count the number of non-unique rows, first notice that a nonzero row can only be identical to the row exactly above or below. Suppose now that a non-zero row i is identical to the row $(i-1)$ above. Then α_{i-1} has to be 0 while α_i and α_{i-2} have to be 1. But then row i cannot simultaneously be identical to row $(i+1)$ below. Thus it is sufficient to calculate the number of non-zero rows minus the number of rows that are identical to the row above it to find the rank of the matrix. Noting that row i is non-zero iff $\alpha_i \alpha_{i-1}$ and that $\alpha_i \overline{\alpha_{i-1}} \alpha_{i-2}$ is only equal 1 when row i is non-zero and equal to the row above it. Thus calculating the number of i for which

$$\alpha_i \alpha_{i-1} + \alpha_i \overline{\alpha_{i-1}} \alpha_{i-2}$$

is equal 1 gives us the rank of L_α. This corresponds to calculating wt(varibits + doublebits).

For the second part of the proof, we need to prove the conditions that check whether $\beta + \alpha \odot S^1(\alpha) \in \mathrm{Img}(L_\alpha)$. First notice that $\alpha \odot S^1(\alpha)$ is in the image of L_α (consider for x the vector with bits alternately set to 0 and 1). Thus it is sufficient to test whether β is in $\mathrm{Img}\, L_\alpha$. Let $y = L_\alpha(x)$. Let us first look at the the case of $\alpha = \mathbf{1}$. Then $L_\alpha(x) = x + S^1(x)$. We can thus deduce from bit y_i whether $x_i = x_{i-1}$ or $x_i \neq x_{i-1}$. Thus the bits in y create a chain of equalities/inequalities in the bits of x which can only be fulfilled if there the number of inequalities is even. Hence in that case $\beta \in \mathrm{Img}\, L_\alpha$ iff $\mathrm{wt}(\beta) \equiv 0 \mod 2$.

For the case that $\alpha \neq \mathbf{1}$, we first note that y_i has to be zero if the corresponding row i in the matrix of Equation 3 is all zeroes. Furthermore following our discussion of this matrix earlier, we see that y_i is independent of the rest of y if the corresponding row is linearly independent of the other rows and that y_i has to be the same as y_{i-1} if the corresponding rows are identical. Thus we only need to check that the zero-rows of the matrix correspond to zero bits in β and that the bits in β which correspond to identical rows in the matrix are equal. Thus β is in the image of L_α iff $\beta \odot \overline{\texttt{varibits}} = \mathbf{0}$ and $(\beta + S^1(\beta)) \odot \texttt{doublebits} = \mathbf{0}$. □

3.2 The full formula for differentials.

Above we treated only the case for the simplified function $f(x) = x \cdot S^1(x)$. As mentioned earlier, the general case where $\gcd(a - b, n) = 1$ can be deduced from this with linear algebra. When $\gcd(d, n) \neq 1$ though, the function $f(x) = x \odot S^d(x)$ partitions the output bits into independent classes. This not only raises differential probabilities (worst case $d = 0$), it also makes the notation for the formulas more complex and cumbersome, though not difficult. We thus restrict ourselves to the most important case when $\gcd(a - b, n) = 1$. The general formulas are then

Theorem 4. *Let* $f(x) = S^a(x) \odot S^b(x) + S^c(x)$, *where* $\gcd(n, a - b) = 1$, n *even, and* $a > b$ *and let* α *and* β *be an input and an output difference. Then with*

$$\texttt{varibits} = S^a(\alpha) \vee S^b(\alpha)$$

and

$$\texttt{doublebits} = S^b(\alpha) \odot \overline{S^a(\alpha)} \odot S^{2a-b}(\alpha)$$

and

$$\gamma = \beta + S^c(\alpha)$$

we have that the probability that difference α goes to difference β is

$$P(\alpha \to \beta) = \begin{cases} 2^{-n+1} & \text{if } \alpha = \mathbf{1} \text{ and } \mathrm{wt}(\gamma) \equiv 0 \mod 2 \\ 2^{-\mathrm{wt}(\mathtt{varibits}+\mathtt{doublebits})} & \text{if } \alpha \neq \mathbf{1} \text{ and } \gamma \odot \overline{\mathtt{varibits}} = \mathbf{0} \\ & \text{and } (\gamma + S^{a-b}(\gamma)) \odot \mathtt{doublebits} = \mathbf{0} \\ 0 & \text{else .} \end{cases}$$

For a more intuitive approach and some elaboration on the differential probabilities, we refer to the ePrint version of this paper.

4 Linear Correlations of SIMON-like round functions

As in the differential case, for the study of linear approximations, we also build up on the results from Section 2.3 and Section 2.4. We will thus start with studying linear approximations for the function $f(x) = x \odot S^a(x)$. Again, the key point here is that all those functions are quadratic and thus their Fourier coefficient, or equivalently their correlation, can be computed by linear algebra (cf. Theorem 5). Theorem 6 is then, in analogy to the differential case, the explicit expression for the linear correlations. It basically corresponds to an explicit formula for the dimension of the involved subspace.

The first result is the following:

Theorem 5.

$$\widehat{f}(\alpha,\beta)^2 = \begin{cases} 2^{n+d} & \text{if } \alpha \in U_\beta^\perp \\ 0 & \text{else} \end{cases}$$

where

$$d = \dim U_\beta$$

and

$$U_\beta = \{y \mid \beta \odot S^a(y) + S^{-a}(\beta \odot y) = \mathbf{0}\}$$

Proof. We compute

$$\begin{aligned}
\widehat{f}(\alpha,\beta)^2 &= \sum_{x,y} \mu\left(\langle\beta, f(x)+f(y)\rangle + \langle\alpha, x+y\rangle\right) \\
&= \sum_{x,y} \mu\left(\langle\beta, f(x)+f(x+y)\rangle + \langle\alpha, y\rangle\right) \\
&= \sum_{x,y} \mu\left(\langle\beta, x\odot S^a(x)+(x+y)\odot S^a(x+y)\rangle + \langle\alpha, y\rangle\right) \\
&= \sum_{y} \mu\left(\langle\beta, f(y)\rangle + \langle\alpha, y\rangle\right) \sum_{x} \mu\left(\langle\beta, x\odot S^a(y)+y\odot S^a(x)\rangle\right) \\
&= \sum_{y} \mu\left(\langle\beta, f(y)\rangle + \langle\alpha, y\rangle\right) \sum_{x} \mu\left(\langle x, \beta\odot S^a(y)+S^{-a}(\beta\odot y)\rangle\right) \quad .
\end{aligned}$$

Now for the sum over x only two outcomes are possible, 2^n or zero. More precisely, it holds that

$$\sum_{x} \mu\left(\langle x, \beta\odot S^a(y)+S^{-a}(\beta\odot y)\rangle\right) = \begin{cases} 2^n & \text{if } \beta\odot S^a(y)+S^{-a}(\beta\odot y)=\mathbf{0} \\ 0 & \text{else} \quad . \end{cases}$$

Thus, defining

$$U_\beta = \{y \mid \beta\odot S^a(y)+S^{-a}(\beta\odot y)=\mathbf{0}\}$$

we get

$$\widehat{f}(\alpha,\beta)^2 = 2^n \sum_{y\in U_\beta} \mu\left(\langle\beta, f(y)\rangle + \langle\alpha, y\rangle\right).$$

Now as

$$\begin{aligned}
\langle\beta, f(y)\rangle &= \langle\beta, y\odot S^a(y)\rangle && (1)\\
&= \langle\mathbf{1}, y\odot\beta\odot S^a(y)\rangle && (2)\\
&= \langle\mathbf{1}, y\odot S^{-a}(\beta\odot y)\rangle && (3)\\
& && (4)
\end{aligned}$$

Now, the function $f_\beta := \langle\beta, f(y)\rangle$ is linear over U_β as can be easily seen by the definition of U_β. Moreover, as f_β is unbalanced for all β, it follows that actually f_β is constant zero on U_β. We thus conclude that

$$\widehat{f}(\alpha,\beta)^2 = 2^n \sum_{y\in U_\beta} \mu\left(\langle\alpha, y\rangle\right).$$

With a similar argument as above, it follows that $\widehat{f}(\alpha,\beta)^2$ is non-zero if and only if α is contained in $U_\beta^\perp$. □

Let us now restrict ourselves to the case where $f(x) = x \odot S^1(x)$. The general case can be deduced analogously to the differential probabilities. For simplicity we also restrict ourselves to the case where n is even.

First we need to introduce some notation. Let $x \in \mathbb{F}_2^n$ with not all bits equal to 1. We now look at blocks of consecutive 1s in x, including potentially a block that "wraps around" the ends of x. Let the lengths of these blocks, measured in bits, be denoted as $c_0, \ldots, c_m$. For example, the bitstring 100101111011 has blocks of length 1, 3, and 4. With this notation define $\theta(x) := \sum_{i=0}^{m} \lceil \frac{c_i}{2} \rceil$.

Noting that the linear square correlation of f is $\frac{\widehat{f}(\alpha,\beta)^2}{2^{2n}}$, we then have the following theorem:

Theorem 6. *With the notation from above it holds that the linear square correlation of $\alpha \xrightarrow{f} \beta$ can be calculated as*

$$C(\alpha \to \beta) = \begin{cases} 2^{-n+2} & \text{if } \beta = \mathbf{1} \text{ and } \alpha \in U_\beta^\perp \\ 2^{-\theta(\beta))} & \text{if } \beta \neq \mathbf{1} \text{ and } \alpha \in U_\beta^\perp \\ 0 & \text{else.} \end{cases}$$

Proof. Define $L_\beta(x) := \beta \odot S^1(x) + S^{-1}(\beta \odot x)$. Clearly L_β is linear. Also $U_\beta = \ker L_\beta(x)$. Let us determine the rank of this mapping. Define the matrices $M_{\beta\cdot}$, M_{S^1}, and $M_{S^{-1}}$ as

$$M_{\beta\cdot} = \begin{pmatrix} \beta_0 & \ldots & \ldots & 0 \\ \vdots & \beta_1 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \ldots & \ldots & \beta_{n-1} \end{pmatrix} \quad \begin{aligned} M_{S^1} &= \begin{pmatrix} 0_{1,n-1} & I_{1,1} \\ I_{n-1,n-1} & 0_{n-1,1} \end{pmatrix} \\ M_{S^{-1}} &= \begin{pmatrix} 0_{n-1,1} & I_{n-1,n-1} \\ I_{1,1} & 0_{1,n-1} \end{pmatrix} \end{aligned} \tag{5}$$

We can then write L_β in matrix form as

$$\begin{pmatrix} 0 & \beta_1 & 0 & \ldots & 0 & \beta_0 \\ \beta_1 & 0 & \beta_2 & 0 & \ldots & 0 \\ 0 & \beta_2 & 0 & \beta_3 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & \ddots & 0 & \beta_{n-1} \\ \beta_0 & 0 & \ldots & 0 & \beta_{n-1} & 0 \end{pmatrix} \tag{6}$$

Clearly, if β is all 1s, the rank of the matrix is $n-2$ as n is even.[4] Let us therefore now assume that β is not all 1s. When we look at a block of 1s in

[4] The rank is $n-1$ when n is odd.

β e.g., $\beta_{i-1} = 0$, $\beta_i, \beta_{i+1}, \ldots, \beta_{i+l-1} = 1$, and $\beta_l = 0$. Then clearly the l rows $i, i+1, \ldots, i+l-1$ are linearly independent when l is even. When l is odd though, the sum of rows i, $i+2$, $i+4$, up to row $i+l-3$ will equal row $i+l-1$. In that case there are thus only $l-1$ linearly independent rows. As the blocks of 1s in β generate independent blocks of rows, we can summarise that the rank of the matrix is exactly $\theta(\beta)$. □

Analogously to the differential probabilities, the linear probabilities in the general case can be derived from this. It is likewise straightforward to derive how to determine whether $\alpha \in U_\beta^\perp$. As an explicit formulation of this is rather tedious, we instead refer to the implementation in Python given in the Section B where both is achieved in the case where $\gcd(a-b, n) = 1$ and n is even.

For a more intuitive approach and some elaboration on the linear probabilities, we refer to the ePrint version of this paper.

5 Finding Optimal Differential and Linear Characteristics

While there are various methods for finding good characteristics, determining optimal differential or linear characteristics remains a hard problem in general. The formulas derived for both differential and linear probabilities enable us to apply an algebraic approach to finding the best characteristics. A similar technique has been applied to the ARX cipher Salsa20 [14] and the CAESAR candidate NORX [4]. For finding the optimal characteristics for Simon we implemented an open source tool [13] based on the SAT/SMT solvers CryptoMiniSat [15] and STP [11].

In the next section we will show how Simon can be modelled to find both the best differential and linear characteristics in this framework and how this can be used to solve cryptanalytic problems.

5.1 Model for Differential Cryptanalysis of SIMON

First we define the variables used in the model of Simon. We use two n-bit variables x_i, y_i to represent the XOR-difference in the left and right halves of the state for each round and an additional variable z_i to store the XOR-difference of the output of the AND operation.

For representing the $\log_2$ probability of the characteristic we introduce an additional variable w_i for each round. The sum over all probabilities w_i then gives the probability of the corresponding differential characteristic. The values w_i are computed according to Theorem 4 as

$$w_i = \mathrm{wt}(\texttt{varibits} + \texttt{doublebits}) \tag{1}$$

where $\mathrm{wt}(x)$ is the Hamming weight of x and

$$\begin{aligned}\texttt{varibits} &= S^a(x_i) \vee S^b(x_i)\\ \texttt{doublebits} &= S^b(x_i) \odot \overline{S^a(x_i)} \wedge S^{2a-b}(x_i)\end{aligned}$$

Therefore, for one round of Simon we get the following set of constraints:

$$\begin{aligned} y_{i+1} &= x_i\\ 0 &= (z_i \odot \texttt{varibits})\\ 0 &= (z_i + S^{a-b}(z_i)) \odot \texttt{doublebits}\\ x_{i+1} &= y_i + z_i + S^c(x_i)\\ w_i &= \mathrm{wt}(\texttt{varibits} + \texttt{doublebits}) \end{aligned} \tag{2}$$

A model for linear characteristics, though slightly more complex, can be implemented in a similar way. A description of this model can be found in the implementation of our framework. Despite the increase in complexity, we could not observe any significant impact on the solving time for the linear model.

5.2 Finding Optimal Characteristics

We can now use the previous model for Simon to search for optimal differential characteristics. This is done by formulating the problem of finding a valid characteristic, with respect to our constraints, for a given probability w. This is important to limit the search space and makes sense as we are usually more interested in differential characteristics with a high probability as they are more promising to lead to attacks with a lower complexity. Therefore, we start with a high probability and check if such a characteristic exists. If not we lower the probability.

The procedure can be described in the following way:

- For each round of the cipher add the corresponding constraints as defined in Equation 2. This system of constraints then exactly describes the form of a valid characteristic for the given parameters.
- Add a condition which accumulates the probabilities of each round as defined in Equation 1 and check if it is equal to our target probability w.
- Query if there exists an assignment of variables which is satisfiable under the constraints.

Table 1: Overview of the optimal differential (on top) and linear characteristics for different variants of SIMON. The probabilities are given as $\log_2(p)$, for linear characteristic the squared correlation is used.

Rounds:	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Differential															
SIMON32	−2	−4	−6	−8	−12	−14	−18	−20	−25	−30	−34	−36	−38	−40	−42
SIMON48	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−35	−38	−44	−46	−50
SIMON64	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−36	−38	−44	−48	−54
Linear															
SIMON32	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−34	−36	−38	−40	−42
SIMON48	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−36	−38	−44	−46	−50
SIMON64	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−36	−38	−44	−48	−54

- Decrement the probability w and repeat the procedure.

One of the main advantages compared to other approaches is that we can prove an upper bound on the probability of characteristics for a given cipher and number of rounds. If the solvers determines the set of conditions unsatisfiable, we know that no characteristic with the specified probability exists. We used this approach to determine the characteristics with optimal probability for different variants of SIMON. The results are given in Table 1.

Upper Bound for the Characteristics.

During our experiments we observed that it seems to be an easy problem for the SMT/SAT solver to prove the absence of differential characteristics above w_{max}. This can be used to get a lower bound on the probability of characteristics contributing to the differential. The procedure is similar to finding the optimal characteristics.

- Start with a very low initial probability w_i.
- Add the same system of constraints which were used for finding the characteristic.
- Add a constraint fixing the variables (x_0, y_0) to Δ_{in} and (x_r, y_r) to Δ_{out}.
- Query if there is a solution for this weight.
- Increase the probability w_i and repeat the procedure until a solution is found.

5.3 Computing the Probability of a Differential

Given a differential characteristic it is of interest to determine the probability of the associated differential $\Pr(\Delta_{in} \xrightarrow{f^r} \Delta_{out})$ as it might potentially have a much higher probability then the single characteristic. It is often assumed

that the probability of the best characteristic can be used to approximate the probability of the best differential. However, this assumption only gives an inaccurate estimate in the case of SIMON.

Similarly to the previous approach for finding the characteristic, we can formalise the problem of finding the probability of a given differential in the following way:

- Add the same system of constraints which were used for finding the characteristic.
- Add a constraint fixing the variables (x_0, y_0) to Δ_{in} and (x_r, y_r) to Δ_{out}.
- Use a SAT solver to find **all** solutions s_i for the probability w.
- Decrement the probability w and repeat the procedure.

The probability of the differential is then given by

$$\Pr(\Delta_{in} \xrightarrow{f^r} \Delta_{out}) = \sum_{i=w_{min}}^{w_{max}} s_i \cdot 2^{-i} \tag{3}$$

where s_i is the number of characteristics with a probability of 2^{-i}.

We used this approach to compute better estimates for the probability of various differentials (see Table 2). In the case of SIMON32 we were able to find *all* characteristics contributing to the differentials for 13 and 14 rounds. The distribution of the characteristics and accumulated probability of the differential is given in Figure 1. It is interesting to see that the distribution of w in the range $[55, 89]$ is close to uniform and therefore the probability of the corresponding differential improves only negligible and converges quickly towards the measured probability[5].

The performance of the whole process is very competitive compared to dedicated approaches. Enumerating all characteristics up to probability 2^{-46} for the 13-round SIMON32 differential takes around 90 seconds on a single CPU core and already gives a better estimate compared to the results in [6]. A complete enumeration of all characteristics for 13-round SIMON32 took close to one core month using CryptoMiniSat4 [15]. The computational effort for other variants of SIMON is comparable given the same number of rounds. However, for these variants we can use differentials with a lower probability covering more rounds due to the increased block size. In this case the running time increases due to the larger interval $[w_{min}, w_{max}]$ and higher number of rounds.

[5] We encrypted all 2^{32} possible texts under 100 random keys to obtain the estimate of the probability for 13-round SIMON32.

For Simon48 and Simon64 we are able to improve the estimate given in [16]. Additionally we found differentials which can cover 17 rounds for Simon48 and 22 rounds for Simon64 which might have potential to improve previous attacks. Our results are also closer to the experimentally obtained estimates given in [10] but give a slightly lower probability. This can be due to the limited number of characteristics we use for the larger Simon variants or the different assumptions on the independence of rounds.

Our results are limited by the available computing power and in general it seems to be difficult to count all characteristics for weights in $[w_{\min}, w_{\max}]$, especially for the larger variants of Simon. However the whole process is embarrassingly parallel, as one can split up the computation for each probability w_i. Furthermore, the improvement that one gets decreases quickly. For all differentials we observed that the distribution of differential characteristics becomes flat after a certain point.

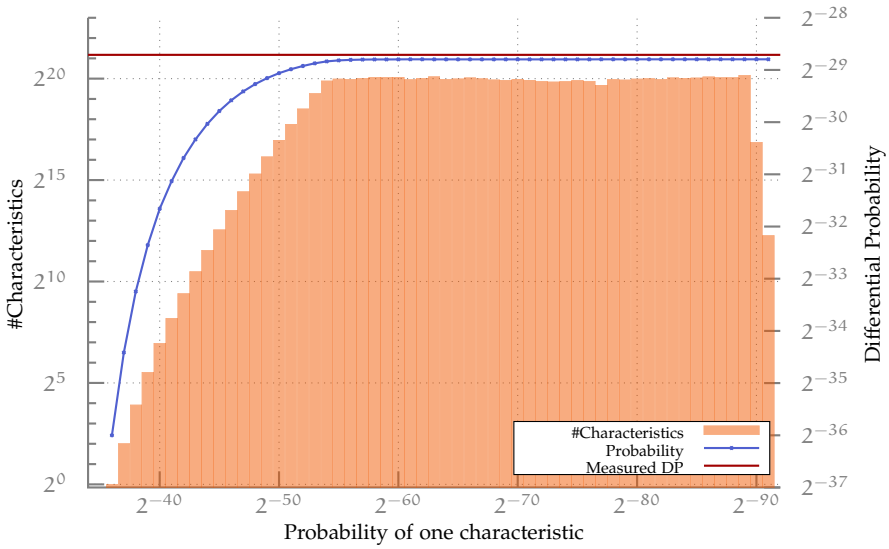


Figure 1: Distribution of the number of characteristics for the differential $(0, 40) \rightarrow (4000, 0)$ for 13-round Simon32 and the accumulated probability. A total of $\approx 2^{25.21}$ characteristics contribute to the probability.

Table 2: Overview of the differentials and the range $[w_{min}, w_{max}]$ of the $\log_2$ probabilities of the characteristics contributing to the differential. For computing the lower bound $\log_2(p)$ of the probability of the differentials, we used all characteristics with probabilities in the range from w_{min} up to the values in brackets in the w_{max} column.

Cipher	Rounds	Δ_{in}	Δ_{out}	w_{min}	w_{max}	$\log_2(p)$
SIMON32	13	(0, 40)	(4000, 0)	36	91 (91)	−28.79
SIMON32	14	(0, 8)	(800, 0)	38	120 (120)	−30.81
SIMON48	15	(20, 800088)	(800208, 2)	46	219 (79)	−41.02
SIMON48	16	(800000, 220082)	(800000, 220000)	50	256 (68)	−44.33
SIMON48	17	(80, 222)	(222, 80)	52	269 (85)	−46.32
SIMON64	21	(4000000, 11000000)	(11000000, 4000000)	68	453 (89)	−57.57
SIMON64	22	(440, 1880)	(440, 100)	72	502 (106)	−61.32

6 Analysis of the Parameter Choices

The designers of SIMON so far gave no justification for their choice of the rotation constants. Here we evaluate the space of rotation parameters with regard to different metrics for the quality of the parameters. Our results are certainly not a definite answer but are rather intended as a starting point to evaluating the design space and reverse engineering the design choices. We consider all possible sets of rotation constants (a, b, c)[6] and checked them for diffusion properties and the optimal differential and linear characteristics.

6.1 Diffusion

As a very simple measure to estimate the quality of the rotation constants, we measure the number of rounds that are needed to reach full diffusion. Full diffusion is reached when every state bit principally depends on all input bits. Compared to computing linear and differential properties it is an easy task to determine the dependency.

In Table 3 we give a comparison to how well the standard SIMON rotation parameters fare within the distribution of all possible parameter sets. The exact distributions for all SIMON variants can be found in the appendix in Table 8.

6.2 Differential and Linear

As a second criteria for our parameters, we computed for all $a > b$ and $\gcd(a - b, n) = 1$ the optimal differential and linear characteristics for 10 rounds of SIMON32, SIMON48 and SIMON64. A list of the parameters which are optimal for all three variants of SIMON can be found in Section D.

[6] Without lack of generality, we assume though that $a \geqslant b$.

Table 3: The number of rounds after which full diffusion is reached for the standard SIMON parameters in comparison to the whole possible set of parameters.

Block size	32	48	64	96	128
Standard parameters	7	8	9	11	13
Median	8	10	11	13	14
First quartile	7	9	9	11	12
Best possible	6	7	8	9	10
Rank	2nd	2nd	2nd	3rd	4th

It is important here to note that there are also many parameter sets, including the standard choice, for which the best 10-round characteristics of SIMON32 have a probability of 2^{-25} compared to the optimum of 2^{-26}. However, this difference by a factor of 2 does not seem to occur for more than 10 rounds and also not any larger variants of SIMON.

6.3 Interesting Alternative Parameter Sets

As one result of our investigation we chose three exemplary sets of parameters that surpass the standard parameters with regards to some metrics. Those variants are SIMON$[12, 5, 3]$, SIMON$[7, 0, 2]$ and SIMON$[1, 0, 2]$.

SIMON$[12, 5, 3]$ has the best diffusion amongst the parameters which have optimal differential and linear characteristics for 10 rounds. The two other choices are both restricted by setting $b = 0$ as this would allow a more efficient implementation in software. Among those SIMON$[7, 0, 2]$ has the best diffusion and the characteristics behave similar to the standard parameters. Ignoring the diffusion SIMON$[1, 0, 2]$ seems also an interesting choice as it is optimal for the differential and linear characteristics.

If we look though at the differential corresponding to the best differential characteristic of SIMON$[7, 0, 2]$ and SIMON$[1, 0, 2]$, then we can see the number of characteristics contributing to it is significantly higher than for the standard parameters (see Table 6). However, for SIMON$[12, 5, 3]$ the differential shows a surprisingly different behaviour and the probability of the differential is much closer to the probability of the characteristic. On the other side, the characteristics seem to be worse for the larger variants as can be seen in Table 7. Furthermore it might be desirable to have at least one rotation parameter that corresponds to a byte length, something that the standard parameter set features.

7 Conclusion and Future Work

In this work we analysed the general class of functions underlying the Simon block cipher. First we rigorously derived efficiently computable and easily implementable expressions for the exact differential and linear behaviour of Simon-like round functions.

Building upon this, we used those expressions for a computer aided approach based on SAT/SMT solvers to find both optimal differential and linear characteristics for Simon. Furthermore, we were able to find all characteristics contributing to the probability of a differential for Simon32 and gave better estimates for the probability for other variants.

Finally, we investigated the space of Simon variants using different rotation constants with respect to diffusion, and the optimal differential and linear characteristics. Interestingly, the default parameters seem to be not always optimal.

This work opens up for further investigations. In particular, the choice and justifications of the NSA parameters for Simon remains unclear. Besides our first progress concerning the round function, the design of the key schedule remains largely unclear and further investigation is needed here.

Acknowledgements

First of all, we wish to thank Tomer Ashur. Both the method to check whether a linear input mask gives a correlated or uncorrelated linear 1-round characteristic for a given output mask as well as the first version of the SMT/SAT model for linear characteristics in Simon were an outcome of our discussions. We furthermore wish to thank the reviewers for comments that helped to improve the paper.

References

[1] Farzaneh Abed, Eik List, Stefan Lucks, and Jakob Wenzel. "Differential Cryptanalysis of Round-Reduced SIMON and SPECK." In: *Fast Software Encryption, FSE 2014*. Ed. by Carlos Cid and Christian Rechberger. Vol. 8540. Lecture Notes in Computer Science. Springer, 2015, pp. 525–545. ISBN: 978-3-662-46705-3.

[2] Martin R. Albrecht, Benedikt Driessen, Elif Bilge Kavun, Gregor Leander, Christof Paar, and Tolga Yalçin. "Block Ciphers - Focus on the Linear Layer (feat. PRIDE)." In: *Advances in Cryptology - CRYPTO 2014*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. Lecture Notes in Computer Science. Springer, 2014, pp. 57–76. ISBN: 978-3-662-44370-5.

[3] Javad Alizadeh, Hoda AlKhzaimi, Mohammad Reza Aref, Nasour Bagheri, Praveen Gauravaram, Abhishek Kumar, Martin M. Lauridsen, and Somitra Kumar Sanadhya. "Cryptanalysis of SIMON Variants with Connections." In: *Radio Frequency Identification: Security and Privacy Issues, RFIDSec 2014*. Ed. by Nitesh Saxena and Ahmad-Reza Sadeghi. Vol. 8651. Lecture Notes in Computer Science. Springer, 2014, pp. 90–107. ISBN: 978-3-319-13065-1.

[4] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. "Analysis of NORX: Investigating Differential and Rotational Properties." In: *Progress in Cryptology - LATINCRYPT 2014*. Ed. by Diego F. Aranha and Alfred Menezes. Vol. 8895. Lecture Notes in Computer Science. Springer, 2015, pp. 306–324. ISBN: 978-3-319-16294-2.

[5] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. *The SIMON and SPECK Families of Lightweight Block Ciphers*. Cryptology ePrint Archive, Report 2013/404. http://eprint.iacr.org/. 2013.

[6] Alex Biryukov, Arnab Roy, and Vesselin Velichkov. "Differential Analysis of Block Ciphers SIMON and SPECK." In: *Fast Software Encryption, FSE 2014*. Ed. by Carlos Cid and Christian Rechberger. Vol. 8540. Lecture Notes in Computer Science. Springer, 2015, pp. 546–570. ISBN: 978-3-662-46705-3.

[7] Julia Borghoff et al. "PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract." In: *Advances in Cryptology - ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Springer, 2012, pp. 208–225. ISBN: 978-3-642-34960-7.

[8] Claude Carlet. "Vectorial Boolean Functions for Cryptography." In: *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*. Vol. 134. Encyclopedia of Mathematics And Its Applications. Cambridge Univ. Press, 2010, pp. 398–469.

[9] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. *The NOEKEON Block Cipher*. Submission to the NESSIE project. 2000.

[10] Itai Dinur, Orr Dunkelman, Masha Gutman, and Adi Shamir. *Improved Top-Down Techniques in Differential Cryptanalysis*. Cryptology ePrint Archive, Report 2015/268. http://eprint.iacr.org/. 2015.

[11] Vijay Ganesh, Trevor Hansen, Mate Soos, Dan Liew, and Ryan Govostes. *STP constraint solver*. https://github.com/stp/stp. 2014.

[12] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. "LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations." In: *Fast Software Encryption, FSE 2014*. Ed. by Carlos Cid and Christian Rechberger. Vol. 8540. Lecture Notes in Computer Science. Springer, 2015, pp. 18–37. ISBN: 978-3-662-46705-3.

[13] Stefan Kölbl. *CryptoSMT: An easy to use tool for cryptanalysis of symmetric primitives*. https://github.com/kste/cryptosmt. 2015.

[14] Nicky Mouha and Bart Preneel. *Towards Finding Optimal Differential Characteristics for ARX: Application to Salsa20*. Cryptology ePrint Archive, Report 2013/328. http://eprint.iacr.org/. 2013.

[15] Mate Soos. *CryptoMiniSat SAT solver*. https://github.com/msoos/cryptominisat/. 2014.

[16] Siwei Sun, Lei Hu, Meiqin Wang, Peng Wang, Kexin Qiao, Xiaoshuang Ma, Danping Shi, Ling Song, and Kai Fu. *Towards Finding the Best Characteristics of Some Bit-oriented Block Ciphers and Automatic Enumeration of (Related-key) Differential and Linear Characteristics with Predefined Properties*. Cryptology ePrint Archive, Report 2014/747. http://eprint.iacr.org/. 2014.

[17] Siwei Sun, Lei Hu, Meiqin Wang, Peng Wang, Kexin Qiao, Xiaoshuang Ma, Danping Shi, Ling Song, and Kai Fu. *Constructing Mixed-integer Programming Models whose Feasible Region is Exactly the Set of All Valid Differential Characteristics of SIMON*. Cryptology ePrint Archive, Report 2015/122. http://eprint.iacr.org/. 2015.

[18] Siwei Sun, Lei Hu, Peng Wang, Kexin Qiao, Xiaoshuang Ma, and Ling Song. "Automatic Security Evaluation and (Related-key) Differential Characteristic Search: Application to SIMON, PRESENT, LBlock, DES(L) and Other Bit-Oriented Block Ciphers." In: *Advances in Cryptology - ASIACRYPT 2014*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science. Springer, 2014, pp. 158–178. ISBN: 978-3-662-45610-1.

[19] Qingju Wang, Zhiqiang Liu, Kerem Varici, Yu Sasaki, Vincent Rijmen, and Yosuke Todo. "Cryptanalysis of Reduced-Round SIMON32 and SIMON48." In: *Progress in Cryptology - INDOCRYPT 2014*. Ed. by Willi Meier and Debdeep Mukhopadhyay. Vol. 8885. Lecture Notes in Computer Science. Springer, 2014, pp. 143–160. ISBN: 978-3-319-13038-5.

A Short tutorial for calculating differential probabilities and square correlations in SIMON-like round functions

The idea of this section is to complement the rigorous proofs with a more intuitive approach to calculating the differential probabilities and square correlation of one round of SIMON. This should also allow us to better understand the Python code given later for calculating these values. We restrict ourselves to a simplified version of the SIMON round function:

$$f : \mathbb{F}_2^n \to \mathbb{F}_2^n \tag{1}$$

$$f(m) = S^1(m) \odot m \ . \tag{2}$$

Writing this equation bitwise where m_i denotes the ith bit of m we obtain:

$$f_i(m) = m_{i-1} \odot m_i \ . \tag{3}$$

When using a bit subscript, we will always implicitly assume that the subscript is calculated modulo n, the number of bits. Thus m_{-1} and m_{n-1} will for example refer to the same bit.

A.1 Differential probabilities

Suppose now we are given a message $m = (m_{n-1}, \ldots, m_1, m_0)$ and an input difference $d = (d_{n-1}, \ldots, d_1, d_0)$. Then the resulting difference D for the function f is calculated as $D(m, d) = f(m) \oplus f(m \oplus d)$. This can be written bitwise as:

$$D_i(m, d) = (m_{i-1} \odot m_i) \oplus ((m_{i-1} \oplus d_{i-1}) \odot (m_i \oplus d_i)) \ . \tag{4}$$

By differentiating between the four possible different cases for d_i and d_{i-1}, we obtain the following:

$$D_i(m, d) = \begin{cases} 0, & \text{if } d_i = 0 \text{ and } d_{i-1} = 0 \\ m_i, & \text{if } d_i = 0 \text{ and } d_{i-1} = 1 \\ m_{i-1}, & \text{if } d_i = 1 \text{ and } d_{i-1} = 0 \\ \overline{m_i \oplus m_{i-1}}, & \text{if } d_i = 1 \text{ and } d_{i-1} = 1 \ . \end{cases} \tag{5}$$

In the last case, D_i is 1 exactly when $m_i = m_{i-1}$ and is 0 when $m_i \neq m_{i-1}$.

Let us now look at a first example. Let $n = 6$, and $d = 001010$. We then calculate $D(m, d)$ using the above bitwise definition of D:

i	5	4	3	2	1	0
d	0	0	1	0	1	0
$S^1(d)$	0	1	0	1	0	0
$D(m, d)$	0	m_4	m_2	m_2	m_0	0

. (6)

We can see that the resulting difference depends only on m_4, m_2 and m_0. Thus by adapting these bits appropriately we can generate the following resulting differences:

$$000000, 000010, 001100, 001110, 010000, 010010, 011100, 011110.$$

All these differences then have the same probability of $8/64 = 1/8$. Note that the reuse of a message bit, m_2 in this case, is due to a subsequence 101 in the difference.

Let us take a look at another example. Let again $n = 6$ and now $d = 011010$. Then we can again calculate $D(m, d)$ as

i	5	4	3	2	1	0
d	0	1	1	1	1	0
$S^1(d)$	1	1	1	1	0	0
$D(m, d)$	m_5	$\overline{m_4 \oplus m_3}$	$\overline{m_3 \oplus m_2}$	$\overline{m_2 \oplus m_1}$	m_0	0

. (7)

We can see here that consecutive 1s in the input difference will cause the respective output difference bit to depend on two message bits. Nevertheless are all five non-zero output difference bits independent of each other. Thus 2^5 different output differences are possible, each one with probability $1/32$.

With the observations made above, we can now devise a rule that allows us to determine the probability of a given pair (α, β) of an input difference α and an output difference β. First we calculate the set of `varibits` which is the bits in which the output difference can be non-zero. So output bit β_i is in `varibits` if and only if α_i or α_{i-1} is non-zero:

$$\texttt{varibits} = \alpha \mid S^1(\alpha) \tag{8}$$

where $\mid$ denotes the bitwise or.

Next we have to calculate which of these output difference bits have to be the same because of patterns 101 in the input difference. We do this by calculating the set `doublebits` which is the output difference bits that always have to be the same as the difference bit one position to the right. Thus β_i is in `doublebits` if and only if α_i is 1, α_{i-1} is 0, and α_{i-2} is 1.

$$\texttt{doublebits} = \alpha \odot \overline{S^1(\alpha)} \odot S^2(\alpha) \tag{9}$$

To check whether input difference α can map to output difference β with non-zero probability, we only need to check whether all non-zero bits of β lie in `varibits` and that all bits of β that are in `doublebits` are the same as the bits to their right. The probability of the transition is then determine by the number of output difference bits that can be chosen freely, i.e. the number of bits in `varibits` minus the number of bits in `doublebits`.

Before we write this procedure down, we have to take a look at one special case, namely when all input difference bits are set, e.g. $n = 6$ and $d = 111111$. Then we can again calculate $D(m, d)$ as

i	5	4	3	2	1	0
d	1	1	1	1	1	1
$S^1(d)$	1	1	1	1	1	1
$D(m, d)$	$\overline{m_5 \oplus m_4}$	$\overline{m_4 \oplus m_3}$	$\overline{m_3 \oplus m_2}$	$\overline{m_2 \oplus m_1}$	$\overline{m_1 \oplus m_0}$	$\overline{m_0 \oplus m_4}$

(10)

Although all bits of the output are influenced and all bits of the input take equal influence, there are not 64 possible output differences since by switching all bits of m the output difference does not change.

So which output differences are possible then? By fixing an output difference, we get a sequence of equations of the kind $m_i = m_{i+1}$ or $m_i \neq m_{i+1}$. This creates a closed chain of equations that have to be coherent to be satisfiable. As a 0 in the output difference creates an inequality and a 1 creates an equality, in the end it boils down to the condition that the number of 0s in the output difference has to be even when the input difference only consists of 1s.

Let us now summarise all of this in a method to calculate the probability that a given input difference α is mapped to a given output difference β:

1. Check if α is the difference with all bits set to 1. If that is the case, calculate the number of 0s in β. If this number is even, return probability 2^{-n+1}, otherwise return probability 0. If α is not all 1s, go to next step.

2. Calculate `varibits` as $\texttt{varibits} = \alpha \mid S^1(\alpha)$. Check if β has any bits set to 1 which are not in `varibits`, i.e. check if $\overline{\texttt{varibits}} \odot \beta \neq \mathbf{0}$. Should this be the case, return probability 0. Otherwise continue with next step.

3. Calculate `doublebits` as $\texttt{doublebits} = \alpha \odot \overline{S^1(\alpha)} \odot S^2(\alpha)$. Check whether there are any bits of β in `doublebits` that are not equal to their right neighbour, i.e. check $\left(\beta + S^1(\beta)\right) \odot \texttt{doublebits} \neq \mathbf{0}$. Should this be the case, return probability 0. Otherwise continue with next step.

4. Return probability $2^{-\mathrm{wt}(\texttt{varibits}+\texttt{doublebits})}$.

This method allows us to determine differential probabilities of the function $f(x) = S^1(x) \odot x$. We only have to apply some affine transformation to convert this to a method for calculating the probability of the SIMON round function. A Python implementation of the more general method can be found in Section B.

A.2 Square correlations

Let us now look at how to calculate square correlations for $f(x) = S^1(x) \odot x$.

First we look at the case where the input mask α is all 0s. Let $n = 6$ and let the output mask β be 010110:

α	0	0	0	0	0	0
m	m_5	m_4	m_3	m_2	m_1	m_0
$S^1(m)$	m_4	m_3	m_2	m_1	m_0	m_5
β	0	1	0	1	1	0

. (11)

The resulting expression is then

$$m_4 m_3 + m_2 m_1 + m_1 m_0. \tag{12}$$

Let us look at the first term. It is zero in 3 out of 4 cases. It thus has a correlation of $\frac{1}{2}$ and hence a square correlation of $\frac{1}{4}$.

Let us look at the next two terms $m_2 m_1$ and $m_1 m_0$. First we note that they are not independent as they share the variable m_1. So we cannot calculate the square correlation of the sum of these terms as the product of the square correlations of the single terms. But we can rewrite the sum of these terms as

$$m_2 m_1 + m_1 m_0 = m_1(m_2 + m_0). \tag{13}$$

Now $(m_2 + m_0)$ behaves like a single one bit variable. Therefore the square correlation of $m_1(m_2 + m_0)$ is $\frac{1}{4}$ as well. As $m_4 m_3$ and $m_1(m_2 + m_0)$ do not share any variables, the square correlation of the whole expression $m_4 m_3 + m_2 m_1 + m_1 m_0$ is then $\frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16}$. It is easy to see that different "blocks" of 1s in β that are separated by at least one 0, will generate independent terms. We thus only need to look at the square correlations of the terms generated from these blocks and multiply these to get the final result.

Let us thus look at a longer block of 1s with $\beta = 011111$:

α	0	0	0	0	0	0
m	m_5	m_4	m_3	m_2	m_1	m_0
$S^1(m)$	m_4	m_3	m_2	m_1	m_0	m_5
β	1	1	1	1	1	0

. (14)

The resulting expression is

$$m_5 m_4 + m_4 m_3 + m_3 m_2 + m_2 m_1 + m_1 m_0. \tag{15}$$

By combining the first and the second as well as the third and the fourth term, we get

$$m_4(m_5 + m_3) + m_2(m_3 + m_1) + m_0 m_1. \tag{16}$$

As $(m_5 + m_3)$, $(m_3 + m_1)$, and m_1 are independent of each other, the three terms $m_4(m_5 + m_3)$, $m_2(m_3 + m_1)$, and $m_0 m_1$ are independent and the square correlation of the whole expression is thus $\left(\frac{1}{4}\right)^3 = 2^{-6}$.

At this point we already dare to formulate a rule. The square correlation of the term generated by m consecutive blocks of 1s is $2^{-2\left\lceil \frac{m}{2} \right\rceil}$. As every pair of consecutive single terms can be combined to create one independent term of square correlation 2^{-2}, the total square correlation just depends on the number of terms left after such pairing. And this number is $\left\lceil \frac{m}{2} \right\rceil$.

Let us now consider a non-zero input mask α. Let $\alpha = 010010$ and let $\beta = 010100$:

α	0	1	0	1	1	1
m	m_5	m_4	m_3	m_2	m_1	m_0
$S^1(m)$	m_4	m_3	m_2	m_1	m_0	m_5
β	0	1	0	1	0	0

. (17)

The resulting expression is then

$$m_4 m_3 + m_4 + m_2 m_1 + m_2 + m_1 + m_0. \tag{18}$$

We see that we can combine the first two terms to get a term of square correlation 2^{-2} again: $m_4 m_3 + m_4 = m_4(m_3 + 1)$. Note that if the second term had been m_3 instead, it would have worked too. For the next three terms we can do the same: $m_2 m_1 + m_2 + m_1 = (m_2 + 1)(m_1 + 1) + 1$. Note that the bias of this term is now flipped; the square correlation is nonetheless also 2^{-2}. As the first two terms are independent of the next three terms, the square correlation of the combined first five terms is 2^{-4}. But when looking at the last term m_0, we see that it is independent of all other terms and unbiased. Thus the square correlation of the complete expression is 0.

As a matter of fact, it is easy to see that when for any i the respective bit α_i of the input mask is 1 but both β_i and β_{i+1} are 0, the resulting expression will always be unbiased. Thus we can say that every non-zero bit in the input mask belonging to some block of 1s in the output mask is a necessary condition for the whole expression to be unbiased. Note that every bit in the input mask can at most be associated with one block of 1s in the output mask.

Thus we can evaluate the square correlation of f for an input mask α and an output mask β like this: First we check whether every non-zero bit in the input mask is associated to a block of 1s in the output mask. Is this not the case, we already know that the square correlation is zero. Otherwise we continue to partition the output mask and the input mask into blocks of 1s and their associated input mask bits. For each of these blocks we determine the square correlation of the resulting expression and finally multiply these together to get the total square correlation.

But how do we evaluate a block of 1s with the associated input mask bits in general? In the last example, we saw that for a block of a single 1 in the output mask, the two associated bits of the input mask can take any value; the square correlation remains 2^{-2}.

How about in the case of a block of two 1s? Let us look at the case of $\alpha = 111001$ and let $\beta = 110110$:

α	1	1	1	0	0	1
m	m_5	m_4	m_3	m_2	m_1	m_0
$S^1(m)$	m_4	m_3	m_2	m_1	m_0	m_5
β	1	1	0	1	1	0

. (19)

The resulting expression is

$$m_5m_4 + m_4m_3 + m_5 + m_4 + m_3 + m_2m_1 + m_1m_0 + m_0. \tag{20}$$

Let us first look at the first block of 1s in the output mask β, i.e. at the expression $m_5m_4 + m_4m_3 + m_5 + m_4 + m_3$. Combining the first two terms, we get

$$m_4(m_5 + m_3) + m_5 + m_4 + m_3. \tag{21}$$

We can now combine the first term with m_4 to get

$$m_4(m_5 + m_3 + 1) + m_5 + m_3. \tag{22}$$

Finally we can also incorporate m_5 and m_3 to get

$$(m_4 + 1)(m_5 + m_3 + 1) + 1. \tag{23}$$

The expression thus has a square correlation of 2^{-2}.

Let us look at the expression generated by the second block of 1s in the output mask:

$$m_2m_1 + m_1m_0 + m_0. \tag{24}$$

Combining the first two terms, we get

$$m_1(m_2 + m_0) + m_0. \tag{25}$$

But now we see that the term m_0 is independent of the first term. Thus we are left with a square correlation of 0. Note that the square correlation would also be 0 if the last term were m_2 but not if the last term were m_1 in which case the square correlation would be 2^{-2}.

As a matter of fact, the rule to determine the square correlation of an expression generated by a block of two 1s in the output mask and the associated bits in the input mask is straightforward. There are three associated input mask bits. If and only if both or none of the two outer bits (m_2 and m_0 in the last example) are set to 1, is the expression biased and the square correlation is 2^{-2}.

In fact, for a block of an even number of 1s in the output mask, any combination of associated input bits, will lead to a biased expression with the same square correlation. For a block of an odd number of 1s in the output mask, we need to check the input mask though. There is an odd number of associated bits to this block in the input mask. Let us refer to the first bit and then every second bit as the odd bits, and to the second bit and then every second bit as the even bits (from which direction we count does not matter). The even bits do not have an influence on the square correlation. But the parity of the odd bits determines whether the expression for this block will be unbiased or not. If and only if the parity is even, the expression is biased.

We can summarise a method to calculate the square correlation for a given input mask α and a given output mask β that is not all 1s as follows:

1. Partition the 1s in the output mask into consecutive blocks of 1s. The total square correlation is now the product of the square correlations for each block.

2. For each block calculate the square correlation:
   a) If the block length is odd, this block is always biased and the square correlation is solely determined by its length.
   b) If the block length is even, we need to check the input mask. There is an odd number of bits in the input mask that are associated with this output block. Calculate the XOR of every second bit of these associated bits starting with the first one (such that both outer bits are considered). If this XOR sum is 1, the block is unbiased and thus the whole expression is unbiased. If the XOR sum is 0, the square correlation for this block is determined by its length.

For an implementation of the method to calculate the square correlation in Python, see Section B.

B Python code to calculate differential probabilities and square correlations in SIMON-like round functions

In the following, code for calculating the differential probabilities and square correlations of SIMON-like round functions ($f_{a,b,c}(x) = S^a(x) \odot S^b(x) + S^c(x)$) are given in Python. Restrictions are that the constants need to fulfil $\gcd(a - b, n) = 1$. We assume that the functions $S^d(x)$ and $wt(x)$ have been implemented as well as a function `parity` that calculates the parity $wt(x) \mod 2$ of a bit vector x. a, b, and c have to be defined in the program as well.

The differential probability of $\alpha \xrightarrow{f} \beta$ can then be calculated with the following function:

```
def pdiff (alpha,beta):
    # Use gamma instead of beta to get rid of linear part
    gamma = beta ^ S(alpha,c)
    # Take care of the case where alpha is all 1s
    if alpha == 2**n-1:
        if hw(~gamma)%2 == 0:
            return 2**(n-1)
        else:
            return 0
    # Determine bits that can take a nonzero difference
    varibits = S(alpha, a) | S(alpha,b)
    # Check whether gamma conforms with varibits
    if gamma & ~varibits != 0:
        return 0
    # Determine the bits that are duplicates
    doublebits = S(alpha,2*a-b) & ~S(alpha,a) & S(alpha,b)
    # Check whether the duplicate bits are the same as there counterpart
    if (gamma ^ S(gamma,a-b)) & doublebits != 0:
        return 0
    return 2**(-hw(varibits^doublebits))
```

The squared correlation of $\alpha \xrightarrow{f} \beta$ can be calculated with the following function. Here we assume n to be even, which is relevant for the case where β is all 1s.

```
def plin (alpha,beta):
    # Get rid of linear part of round function
    alpha ^= S(beta,-c)
    # If the input masks uses bits that have corresponding bits
    # in the output mask, the correlation is 0.
    if ((S(beta,-a) | S(beta,-b)) ^ alpha) & alpha != 0:
        return 0
    # Take care of the case where beta is all 1s
    if beta == 2**n-1:
            t, v = alpha, 0
            while t != 0:
                v ^= t & 3
                t >>= 2
            if v != 0:
```

```
            return 0
        else:
            return 2**(-n+2)
# Set in the abits mask the first and then every second bit of each
# block of 1s in the output mask beta. Each corresponds to one
# independent multiplication term, and thus adds a factor of 2^(-2)
# to the square correlation.
# Example: beta = 0111101110110 -> abits = 0101001010100
tmp = beta
abits = beta
while tmp != 0:
    tmp = beta & S(tmp, -(a-b))
    abits ^= tmp
# The sbits correspond to bits one to the right of each block of an
# even number of 1s in the output mask.
# Example: beta = 0111101110110 -> sbits = 0000010000001
sbits = S(beta, -(a-b)) & ~beta & ~S(abits, -(a-b))
# Adopt sbits to correspond to the respective bits in the input
# mask
sbits = S(sbits, -b)
# The pbits are used to check whether the input mask removes the
# bias from one of the output mask blocks. It checks the parity of
# the sum of every second inputmask bit for each block that
# corresponds to a block of an even number of 1s in the output mask.
pbits = 0
while sbits != 0:
    pbits ^= sbits & alpha
    sbits = S(sbits, (a-b)) & S(beta,-b)
    sbits = S(sbits, (a-b))
    pbits = S(pbits, 2*(a-b))
# If the parity is uneven for any one of the blocks, there is no bias.
if pbits != 0:
    return 0
return 2**(-2*hw(abits))
```

C Additional Differential Bounds

In Table 4 resp. Table 5 we give the distributions for the characteristics contributing to a differential up to the bound we computed them.

D Optimal parameters for differential characteristics

The following sets of rotation constants (a, b, c) are optimal for 10 rounds regarding differential characteristics for SIMON32, SIMON48, and SIMON64

$(1,0,2),(1,0,3),(2,1,3),(4,3,5),(5,0,10),(5,0,15),(5,4,3),(7,0,14),(7,6,5)$
$(8,1,3),(8,3,14),(8,7,5),(10,5,15),(11,6,1),(12,1,7),(12,5,3),(12,7,1)$
$(13,0,10),(13,0,7),(13,8,2)$

Table 4: Number of differential characteristics for the differential $(80, 222) \xrightarrow{f^{17}} (222, 80)$ for SIMON48.

$\log_2(p)$	#Characteristics	$\log_2(p)$	#Characteristics
−52	1	−69	20890
−53	6	−70	38837
−54	15	−71	72822
−55	46	−72	133410
−56	100	−73	240790
−57	208	−74	353176
−58	379	−75	279833
−59	685	−76	235071
−60	1067	−77	259029
−61	1607	−78	225836
−62	2255	−79	256135
−63	2839	−80	252193
−64	3476	−81	252654
−65	4088	−82	198784
−66	5032	−83	229843
−67	7063	−84	208757
−68	11481	−85	253112

Table 5: Number of differential characteristics for the differential $(4000000, 11000000) \xrightarrow{f^{21}} (11000000, 4000000)$ for SIMON64.

$\log_2(p)$	#Characteristics	$\log_2(p)$	#Characteristics
−68	2	−83	185709
−69	14	−84	173860
−70	70	−85	171902
−71	276	−86	171302
−72	951	−87	168190
−73	2880	−88	164694
−74	8101	−89	163141
−75	21062	−90	161089
−76	52255	−91	159354
−77	123206	−92	155804
−78	238297	−93	150954
−79	239305	−94	145061
−80	171895	−95	141914
−81	170187	−96	138480
−82	165671	−97	132931

Table 6: Distribution of the characteristics for a 13-round differential for SIMON32 using different set of constants.

$\log_2(p)$	$[8,1,2]$	$[12,5,3]$	$[7,0,2]$	$[1,0,2]$
−36	1	1	4	1
−37	4	2	16	6
−38	15	3	56	27
−39	46	2	144	88
−40	124	1	336	283
−41	288	0	744	822
−42	673	0	1644	2297
−43	1426	0	3420	6006
−44	2973	0	6933	14954
−45	5962	0	13270	34524
−46	11661	1	24436	73972
−47	21916	3	43784	150272
−48	40226	14	76261	292118
−49	72246	32	130068	/
−50	126574	54	218832	/
−51	218516	83	362284	/

Similar to the experiments for the default parameters, we used our framework to evaluate the quality of various rotation constants. In Table 7 we give an overview of the best differential characteristics for variants of SIMON using a different set of rotation constants. Table 6 shows that a carefully chosen set of constants can have a very strong effect on the differentials.

Table 7: Overview of the optimal differential characteristics for SIMON variants.

Rounds:	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Differential $(12,5,3)$															
SIMON32	−2	−4	−6	−8	−12	−14	−18	−20	−26	−28	−34	−36	−42	−44	−47
SIMON48	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−36	−36	−38	−40	−42
SIMON64	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−35	−37	−43	−47	/
Differential $(1,0,2)$															
SIMON32	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−36	−36	−38	−40	−42
SIMON48	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−36	−38	−44	−48	−54
SIMON64	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−36	−38	−44	−48	−54
Differential $(7,0,2)$															
SIMON32	−2	−4	−6	−8	−12	−14	−18	−20	−25	−30	−35	−36	−38	−40	−42
SIMON48	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−35	−38	−44	−48	−53
SIMON64	−2	−4	−6	−8	−12	−14	−18	−20	−26	−30	−36	−38	−44	−48	/

Table 8: For each Simon variant and each possible number of rounds, the number of possible combinations of rotation constants (a, b, c) with $a \geqslant b$ is given that reaches full diffusion.

Simon32

Rounds	6	7	8	9	10	11	17	∞
#(a, b, c)	48	600	528	88	144	128	64	576

Simon48

Rounds	7	8	9	10	11	13	14	15	25	∞
#(a, b, c)	48	1392	1680	792	528	344	144	128	64	2080

Simon64

Rounds	8	9	10	11	12	13	15	17	18	19	33	∞
#(a, b, c)	384	4800	2112	2256	1152	608	512	48	288	256	128	4352

Simon96

Rounds	9	10	11	12	13	14	15	16	17
#(a, b, c)	336	4272	13920	7104	5568	3456	912	1152	800
	19	21	25	26	27	49	∞		
	1568	640	48	288	256	128	16000		

Simon128

Rounds	10	11	12	13	14	15	16	17	18	19	20
#(a, b, c)	768	10944	26112	25536	9024	6912	7488	2496	192	1824	2304
	21	23	24	25	33	34	35	65	∞		
	1792	1024	960	512	96	576	512	256	33792		

A Brief Comparison of Simon and Simeck

Publication Information

Stefan Kölbl and Arnab Roy. "A Brief Comparison of Simon and Simeck." In: *Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016*. 2016

Contribution

- Main author.

Remarks

This publication has been slightly edited to fit the format.

A Brief Comparison of Simon and Simeck

Stefan Kölbl, Arnab Roy
{stek,arroy}@dtu.dk

DTU Compute, Technical University of Denmark, Denmark

Abstract. Simeck is a new lightweight block cipher design based on combining the design principles of the Simon and Speck block cipher. While the design allows a smaller and more efficient hardware implementation, its security margins are not well understood. The lack of design rationals of its predecessors further leaves some uncertainty on the security of Simeck.

In this work we give a short analysis of the impact of the design changes by comparing the upper bounds on the probability of differential and linear trails with Simon. We also give a comparison of the effort of finding those bounds, which surprisingly is significantly lower for Simeck while covering a larger number of rounds at the same time.

Furthermore, we provide new differentials for Simeck which can cover more rounds compared to previous results on Simon and study how to choose good differentials for attacks and show that one can find better differentials by building them from a larger set of trail with initially lower probability.

We also provide experimental results for the differentials for Simon32 and Simeck32 which show that there exist keys for which the probability of the differential is significantly higher than expected.

Based on this we mount key recovery attacks on 19/26/33 rounds of Simeck32/48/64, which also give insights on the reduced key guessing effort due to the different set of rotation constants.

Keywords: SIMON, SIMECK, differential cryptanalysis, block cipher

1 Introduction

Simeck is a family of lightweight block ciphers proposed in CHES'15 by Yang, Zhu, Suder, Aagaard and Gong [13]. The design combines the Simon and Speck block ciphers proposed by NSA [4], which leads to a more compact and efficient implementation in hardware. The block cipher Simon is built by iterating a very simple round function which uses bitwise AND and rotation while the block cipher Speck uses modular addition as non-linear operations. The designers of Simeck chose a different set of rotation constants from Simon to construct the round function.

The efficiency of Simon and Speck on hardware and software platform has a natural appeal to use similar design principles for constructing efficient primitives. The designers of Simon and Speck do not provide rationales for the original choices apart from implementation aspects. These modifications are likely to have an impact on the security margins, which often are already small for lightweight designs and can be a delicate issue. Hence it is important to understand the effect of the parameter change on the security of Simon like design.

The Simon block cipher family has been studied in various paper [1, 2, 5, 9, 10, 12] and the attacks covering the most rounds are based on differential and linear cryptanalysis, which therefore will also be the focus of this work. However very few analyses [7] was done to study the choice of parameters for Simon and Speck and their effect on the security of these block ciphers.

Our Results

In this paper we give a first analysis on the impact of these design changes by comparing the bounds for differential and linear trails with the corresponding variants of Simon. An unexpected advantage for Simeck is, that it takes significantly less time to find those while also covering more rounds (see Table 1). Additionally we investigate strategies to find differentials which have a high probability and are more suitable for efficient attacks.

Surprisingly, we can find differentials with higher probability for Simeck32 by not using the input and output difference from the best differential trails. Furthermore, we also provide new differentials which cover 4 and 5 rounds for Simeck48 and Simeck64 respectively which also have a slightly higher probability compared to previous results on Simon.

We verified the estimated probability with experiments for both Simon32 and Simeck32 to confirm our model and also noticed that for some keys a surprisingly large number of valid pairs can be found.

This is followed by key-recovery attacks for reduced round versions of Simeck (see Table 6). These attacks are similar to previous work [5] done

on Simon and give insight into the lower complexity for the key recovery process for Simeck as we need to guess fewer key bits.

Table 1: A comparison between the number of rounds for which upper bounds on the probability of differential and linear trails exist, the probability of differentials utilized in attacks and the best differential attacks on Simon and Simeck. Results contributed by this work are marked in bold.

Cipher	Rounds	Upper Bounds		Differentials		Key Recovery
		differential	linear	Rounds	$\Pr(\alpha \to \beta)$	
Simon32/64	32	**32**	**32**	13	$2^{-28.79}$ [5]	21 [11]
Simeck32/64	32	**32**	**32**	**13**	$\mathbf{2^{-27.28}}$	22 [8]
Simon48/96	36	**19**	**20**	16	$2^{-44.65}$ [10]	24 [11]
Simeck48/96	36	**36**	**36**	**20**	$\mathbf{2^{-43.65}}$	26 [8]
Simon64/128	44	15 [7]	**17**	21	$2^{-60.21}$ [10]	29 [11]
Simeck64/128	44	**40**	**41**	**26**	$\mathbf{2^{-60.02}}$	35 [8]

2 The Simeck Block Cipher

Simeck2n is a family of block ciphers with n-bit word size, where $n = 16, 24, 32$. Each variant has a block size of $2n$ and key size of $4n$ giving the three variants of Simeck: Simeck32/64, Simeck48/96 and Simeck64/128. As for each block size there is only one key size we will omit the key size usually.

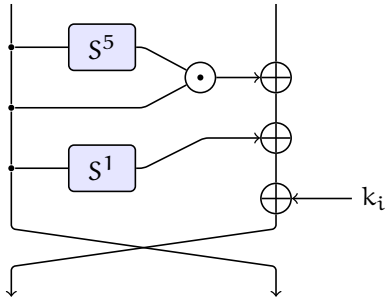


Figure 1: The round function of Simeck.

The block cipher is based on the *Feistel* construction and the round function f is the same as in Simon apart from using $(5, 0, 1)$ for the rotation constants (as depicted in Figure 1). The key-schedule on the other hand is similar to

Speck, reusing the round function to update the keys. The key K is split into four words (t_2, t_1, t_0, k_0) and the round keys $k_0, \ldots, k_{r-1}$ are given by:

$$\begin{aligned} k_{i+1} &= t_i \\ t_{i+3} &= k_i \oplus f(t_i) \oplus C \end{aligned} \tag{1}$$

3 Preliminaries

Differential cryptanalysis is a powerful tool for analyzing block ciphers using a chosen plaintext attack. The idea is to find a correlation between the difference of a pair of plaintexts and the corresponding pair of ciphertexts. Resistance to differential cryptanalysis is an important design criteria but it is difficult, especially for designs like Simon, to proof the resistance against it.

Definition 1. A *differential trail* Q is a sequence of difference patterns

$$Q = (\alpha_0 \xrightarrow{f_0} \alpha_1 \xrightarrow{f_1} \cdots \alpha_{r-1} \xrightarrow{f_{r-1}} \alpha_r). \tag{1}$$

In general, as the key is unknown to an attacker, we are interested in the probability that a random pair of inputs follows such a differential trail and the goal for the attacker is to find a correlation between input and output difference with high probability.

Definition 2. The probability of a differential trail Q is defined as

$$\Pr(\alpha_0 \xrightarrow{f_0} \alpha_1 \xrightarrow{f_1} \cdots \alpha_{r-1} \xrightarrow{f_{r-1}} \alpha_r) = \prod_{t=0}^{r-1} \Pr(\alpha_t \to \alpha_{t+1}) \tag{2}$$

and gives the probability that a random input follows the differential trail. The last equality holds if we assume independent rounds.

In most attack scenarios we are not interested in the probability of a differential trail, as we are only interested in the input difference α_0 and the output difference α_r, but not what happens in between.

Definition 3. The probability of a *differential* is the sum of all r round differential trails

$$\Pr(\alpha_0 \xrightarrow{f} \alpha_r) = \sum_{\alpha_1, \ldots, \alpha_{r-1}} (\alpha_0 \xrightarrow{f_0} \alpha_1 \xrightarrow{f_1} \cdots \alpha_{r-1} \xrightarrow{f_{r-1}} \alpha_r) \tag{3}$$

which have the same input and output difference.

Table 2: Number of rounds required for full diffusion.

Wordsize	32-bit	48-bit	64-bit
Simon	7 Rounds	8 Rounds	9 Rounds
Simeck	8 Rounds	9 Rounds	11 Rounds

4 Analysis of Simon and Simeck

In [7] the differential and linear properties of Simon were studied, including variants using a different set of rotation constants. Following up on this work, we can use the same methods to analyze the round function of Simeck. This allows us to find lower bounds for the probability of a differential trail resp. square correlation of a linear trail for a given number of rounds.

4.1 Diffusion

An important criteria for the quality of a round function in a block cipher is the amount of diffusion it provides, i.e. how many rounds r it takes until each bit at the input effects all bits of the output. For Simon this was already studied in [7] for the whole parameter set and we only explicitly state the comparison to Simeck here in Table 2.

4.2 Bounds on the best differential trails

We carried out experiments for the parameter set of Simeck using CryptoSMT[1] to find the optimal differential and linear trails for Simeck32, Simeck48 and Simeck64 and compare it with the results on Simon. The results of this experiment are given in Figure 1. The bounds on the square correlation for linear trails are given in the Appendix.

While the bounds for Simon32 and Simeck32 are still comparable we noticed a significant difference for the larger variants. While the required number of rounds for Simon48, such that the probability of the best trail is less than 2^{-48}, is 16, Simeck48 achieves the same property only after 20 rounds. It is also interesting to note that the bounds for the different word sizes of Simeck are the same, which is not the case for Simon.

In our experiments we noticed that the different set of rotation constants plays a huge role in the running time of the SMT solver. For instance finding the bounds in Figure 1 took 51 hours for Simon32 and 10 hours for Simeck32[2]. Especially for larger block sizes it allows us to provide bounds for

[1] CryptoSMT https://github.com/kste/cryptosmt Version: 70794d83

[2] Using Boolector 2.0.1. running on an Intel Xeon X5650 2.66GHz 48GB RAM (1 core).

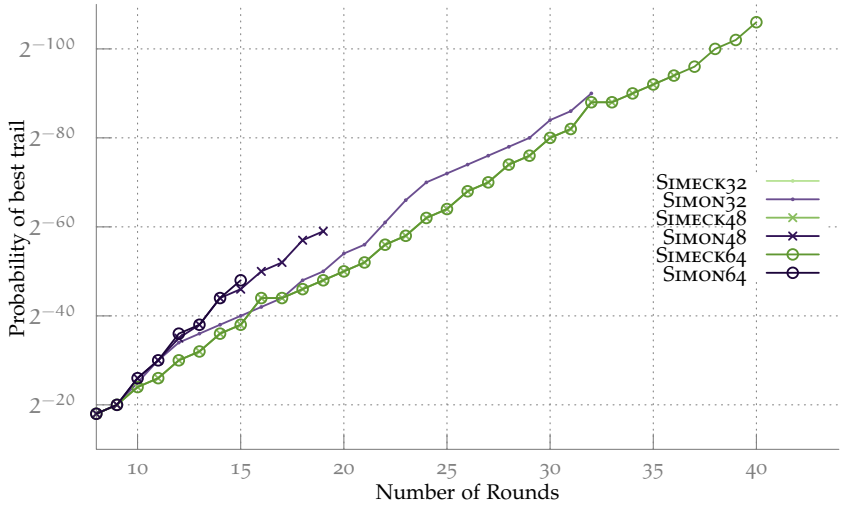


Figure 1: Lower bounds on the probability of the best differential trails for variants of Simon and Simeck. For the different variants of Simeck the bounds are the same.

a significant larger number of rounds including full Simeck48. For Simon64 computing the bounds up to 15 rounds takes around 19 hours, while the same process only takes around 30 minutes for Simeck64. We computed the bounds for Simeck64 up to round 40 in around 53 hours.

4.3 Differential effect in Simon and Simeck

As noted in previous works Simon shows a strong differential resp. linear hull effect, which invalidates an often made assumption that the probability of the best trail can be used to estimate the probability of the best differential. Therefore bounds on differential and linear trails have to be treated with caution. The choice of constants for Simon-like round functions also plays a role in this as shown in [7].

One approach to find good differentials is to first find the best trail for a given number of rounds of Simeck using CryptoSMT [6] and then find a large set of trails with the same input and output difference. However, as we will see later this will not always give the highest probability differential. The results of these experiments are summarized in Table 3.

If we compare those with previous results on Simon we can cover more rounds. The best previous differential attack by Wang, Wang, Jia and Zhao [11] utilizes a 13-round differential for Simon32, a 16-round differ-

ential for Simon48 and a 21-round differential for Simon64. We show that with the same or slightly better probability (Table 1) differentials can be found for a higher number of rounds for both Simeck48 and Simeck64.

Table 3: Overview of the differentials we found for Simeck which can likely be used to mount attacks. The probability is given by summing up all trails up to probability 2^{max} taking a time T.

Cipher	Rounds	$Q = (\alpha \to \beta)$	$\log_2(p)$	max	T
Simeck32	13	$(8000, 4011) \to (4000, 0)$	−27.28	−49	17h
Simeck48	20	$(20000, 450000) \to (30000, 10000)$	−43.65	−98	135h
Simeck48	20	$(400000, e00000) \to (400000, 200000)$	−43.65	−74	93h
Simeck48	21	$(20000, 470000) \to (50000, 20000)$	−45.65	−100	130h
Simeck64	25	$(2, 40000007) \to (40000045, 2)$	−56.78	−90	110h
Simeck64	26	$(0, 4400000) \to (8800000, 400000)$	−60.02	−121	120h

While we let our experiments run for a few days, the probability only improves marginally after a short time. For instance, for Simeck32 and Simeck48 the estimates after three minutes are only 2^{-2} lower than the final results and after two hours the improvements are very small. Some additional details on the differential utilized in the key-recovery attack on Simeck48 can be found in Table 9, including the exact running times to obtain the results.

4.4 Choosing a good differential for attacks

For an attack we want a differential with a high probability, but also the form of the input and output difference can have an influence on the resulting attack complexity. Ideally we want differentials with a sparse input/output difference resp. of the form $(x, 0) \to (0, x)$. When expanding such a differential it leads to a truncated differential with fewer unknown bits which reduces the complexity in the key recovery part of the attack as will be seen later.

The best differential trail of the form $(x, 0) \to (0, x)$ only has a probability of 2^{-42} for Simeck32 resp. 2^{-47} for Simon32. The corresponding differential improves the probability to $\approx 2^{-36.7}$, but is still unlikely to be useful for an attack. If we relax the restriction and allow differentials of the form $(x, x) \to (0, x)$ we can find differential trails with a probability of 2^{-38} (the same bound exists for Simon32). However, the corresponding differentials still seem impractical for an attack. As both this approaches fail for finding good differentials we do not impose any restrictions on the form of the input resp. output difference of the differentials.

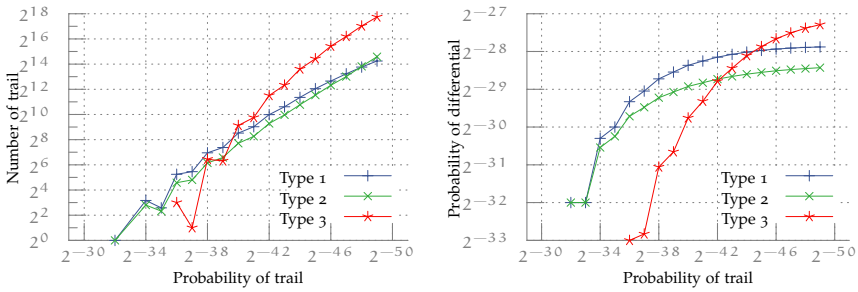
We looked at all 40 rotation invariant differentials constructed from the best differential trail with probability 2^{-32} for Simeck32 (see Table 4). There

Table 4: Number of differential trails for 13-round Simeck32.

$\Pr(\alpha \xrightarrow{f^{13}} \beta)$	Trails
2^{-32}	640
2^{-33}	128
2^{-34}	31616
2^{-35}	49152

are only two possible distributions for the trails contributing to the differential, which we denote as Type 1 and Type 2 (see Figure 2 and Table 8). There are 8 trails of Type 1, all with at least one word having 0 difference, and the corresponding differential gives a slightly higher probability. For a list of these differentials see Table 7.

However, by expanding our search we could find a better differential. Instead of using the optimal differential trail we can find the differential $(8000, 4011) \rightarrow (4000, 0)$ which has a higher probability even though the best trail contributing only has a probability of 2^{-36}. This is due to the higher number of trails contributing to this specific differential (see Type 3 in Figure 2 respectively Table 8).



(a) Number of trails (b) Accumulated Probability

Figure 2: Distribution of trails contributing to the differentials for 13 rounds of Simeck32 and the accumulated probability by summing up all trails up to a specific probability.

For 20-round Simeck48 the best trails with pattern only has a probability of 2^{-62} and for $(x, x) \rightarrow (0, x)$ it is 2^{-54}. The corresponding differentials are not usable for an attack in this case. Therefore, we again do not impose any of these restrictions and use the 20-round trails with highest probability. For Simeck48 there are 768 such trails with a probability of 2^{-50} (32 rotation

invariant) and we choose the one where the input and output difference is most sparse.

For Simeck64 the best differentials we found are also based on the best trail and given in Table 3.

4.5 Experimental Verification

While the previous approach can give a good estimate for the probability one can expect for a differential, it is not entirely clear how good these approximations are. As both Simon32 and Simeck32 allow us to run experiments on the full codebook we can verify the probabilities at least for these variants. For a random function we expect that the number of valid pairs are a Poisson distribution.

Definition 4. Let X be a Poisson distributed random variable representing the number of pairs (a, b) with values in $\mathbb{F}_2^n$ following a differential $Q = (\alpha \xrightarrow{f} \beta)$, that means $f(a) \oplus f(a \oplus \alpha) = \beta$, then

$$\Pr(X = l) = \frac{1}{2}(2^n p)^l \frac{e^{-(2^n p)}}{l!} \tag{1}$$

where p is the probability of the differential.

We ran experiments for both Simon32 and Simeck32 reduced to 13 rounds by encrypting the full code book for a large number of random keys. The differential we used for Simon32 is $(0, 40) \rightarrow (4000, 0)$, which is also used in the best attack so far [11] and has an estimated probability of $2^{-28.56}$. The expected number of valid pairs is $\mathbf{E}(X) \approx 5.425$. We encrypted the full code book using 202225 random master keys and counted the number of unique pairs. The full distribution is given in Figure 3. The distribution follows the model in Equation 1, but we observe some unusual high number of pairs for some keys. For example the key $K = (k_0, k_1, k_2, k_3) = (\texttt{8ec1}, \texttt{1cf8}, \texttt{e84a}, \texttt{cee2})$ gives 1082 pairs following the differential. If 13 rounds of Simon32 would behave like a random function, this would only occur with an extremely low probability $\Pr(X = 1082) \ll 2^{-1000}$.

For Simeck32 we used the new differential $(8000, 4011) \rightarrow (4000, 0)$ with $\mathbf{E}(X) \approx 13.175$. Again, we encrypt the full code book for 134570 random keys and the distribution follows our model as can be seen in Figure 4. Similar, to Simon for some keys a surprisingly large number of valid pairs can be found. In both cases our method provides a good estimate for the probability of a differential and we can use Equation 1 for estimating the number of pairs.

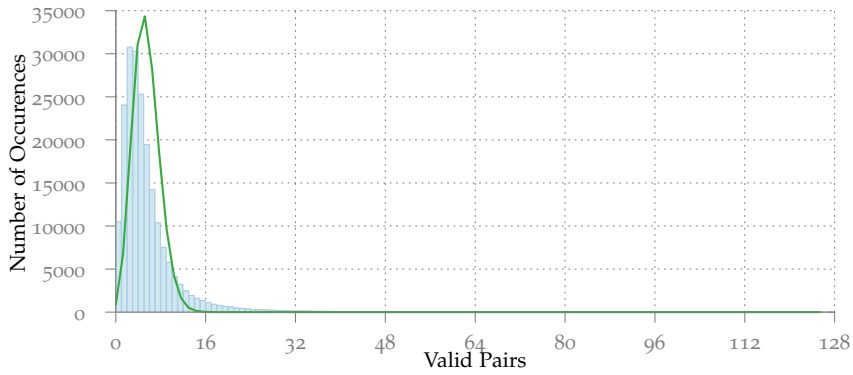


Figure 3: Distribution of how many times we observe l valid pairs for the differential $(0,40) \xrightarrow{f^{13}} (4000,0)$ for Simon32 using a random key.

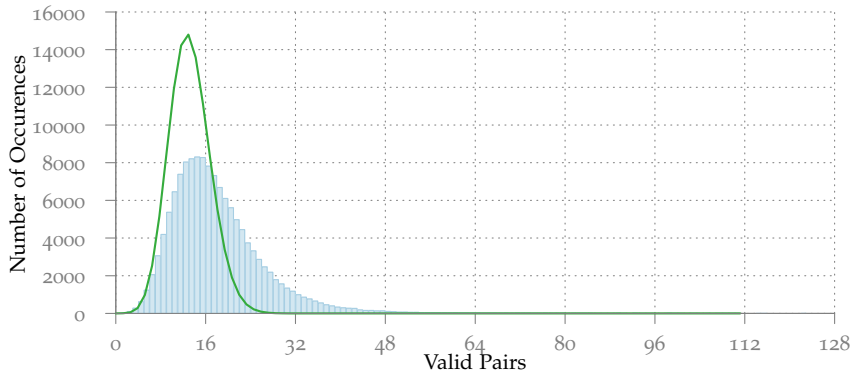


Figure 4: Distribution of how many times we observe l valid pairs for the differential $(8000,4011) \xrightarrow{f^{13}} (4000,0)$ for Simeck32 using a random key.

Table 5: Truncated differential obtained by extending $(400000, e00000) \xrightarrow{20} (400000, 200000)$ in both directions until all bits are unknown.

Round	ΔL	ΔR	*	*
−5	***0***0****************	**********************	22	24
−4	***000000***0**********	***0***0****************	17	22
−3	***00000000000***0****1*	***000000***0**********	11	17
−2	***0000000000000000***01	***00000000000***0****1*	6	11
−1	11100000000000000000000	***0000000000000000***01	0	6
0	01000000000000000000000	11100000000000000000000	0	0
	20 rounds			
20	01000000000000000000000	00100000000000000000000	0	0
21	1*10000000000000000*000	01000000000000000000000	2	0
22	***000000000000*000***01	1*10000000000000000*000	7	2
23	***0000000*000***0****1*	***000000000000*000***01	12	7
24	***00*000***0**********	***0000000*000***0****1*	18	12
25	***0***0****************	***00*000***0**********	22	18
26	**********************	***0***0****************	24	22

5 Recovering the Key

In the following subsection we describe the key recovery attack on Simeck48 based on the differential given in Table 3. Extending this differential both in forward and backward directions gives the truncated differential shown in Table 5 which will be used in the attack. The input difference to round r is denoted as Δ^r and k_r denotes the round key for round r. The difference in the left resp. right part of the state we denote as ΔL^r and ΔR^r.

5.1 Attack on 26-round Simeck48

Our attack on 26-round Simeck48 uses four 20-round differentials in a similar way as in [5]. Let D_i denote the differentials

$$D_1 : (400000,\ e00000) \xrightarrow{f^{20}} (400000,\ 200000)$$
$$D_2 : (800000,\ c00001) \xrightarrow{f^{20}} (800000,\ 400000)$$
$$D_3 : (000004,\ 00000e) \xrightarrow{f^{20}} (000004,\ 000002)$$
$$D_4 : (000008,\ 00001c) \xrightarrow{f^{20}} (000008,\ 000004)$$

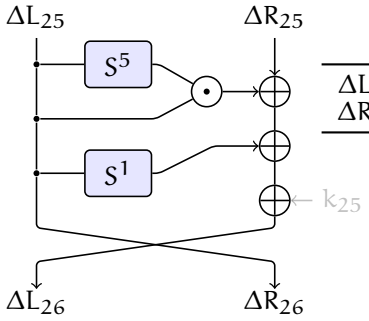
each having probability $\approx 2^{-44}$. We add 4 rounds at the end and 2 rounds on top and obtain the truncated difference (see Table 5). The truncated difference at round 0 for each differential is given by

```
***00000000000000000***01,***000000000000***0****1*
**0000000000000000***01*,**0000000000***0****1**
000000000000000***01***0,000000000***0****1****0
000000000000000***01***00,000000000***0****1****00 .
```

By identifying the unknown and known bit positions in these differentials we can construct a set of 2^{30} plaintext pairs where the bit positions corresponding to the aligned 0s in the truncated differentials are fixed to an arbitrary value for all plain-texts. By guessing 6 round key bits we can also identify the 2^{31} pairs satisfying the difference $(\Delta L^2, \Delta R^2)$ after the first two round encryption. Hence we can get 4 sets of 2^{31} pairs of plain-texts where the difference is satisfied after the first two rounds of encryption. By varying the fixed bit positions we can get 4 sets of 2^{46} pairs of plain-texts, each satisfying the difference after two rounds for each key guess.

Filtering the pairs

First we encrypt the 2^{46} plaintext pairs. Then we unroll the last round and use the truncated differential to verify if a pair is valid. This is possible due to the last key addition not having any influence on the difference $(\Delta L^{25}, \Delta R^{25})$. As there are $12 + 17$ bits known in this round we will have $2^{46-29} = 2^{17}$ plaintext pairs left.



ΔL_{25}	***0000000*000***0****1*	12
ΔR_{25}	***0000000000000*000***01	17

For each pair $(\Delta L', \Delta R')$ obtained:

1. Compute $\Delta R'_{25} = f(\Delta R'_{26}) \oplus \Delta L_{26}$[3].
2. Check if $\Delta L'_{25} = \Delta L_{25}$ and $\Delta R'_{25} = \Delta R_{25}$.

Figure 1: Filtering for the correct pairs which we use in the key guessing part.

Key guessing

In the key guessing phase we guess the necessary round key bits (or linear combination of round key bits) to verify the difference at the beginning of round 22, i.e. Δ^{22}. For each differential we counted that a total of 30 round key bits and linear combinations of round key bits are necessary to be guessed during this process. The required key bits D_1^K for D_1 are

$$K^{23} = \{2, 17\}$$
$$K^{24} = \{2, 3, 4, 8, 12, 16, 17, 18, 22\}$$
$$K^{25} = \{1, 2, 3, 4, 5, 7, 8, 9, 11, 12, 13, 15, 16, 17, 18, 19, 21, 22, 23\}$$

We describe this process for one round in Figure 2. An interesting difference to Simon in the key guessing part is that the required number of key guesses is much lower, as many bits required to guess coincide when partially recovering the state which can reduce the overall complexity. This is always the case if one of the rotation constants is zero, but similar effects can occur with other choices as well.

For the key guessing part, we keep an array of 2^{30} counters and increment a counter when it is correctly verified with the difference after partial decryption of the cipher-text pairs. For each differential we can verify the remaining $19 (= 48 - 29)$ bits with the key guessing process. For the 2^{30} counters we expect to have $(2^{17} \times 2^{30})/2^{19} = 2^{28}$ increments. The probability of a counter being incremented is $2^{28}/2^{30} = 2^{-2}$. Since 4 correct pairs are expected to be among the filtered pairs, the expected number of counters having having at least 4 increments is

$$2^{30} \cdot (1 - \Pr(X < 4)) \approx 2^{17.13}. \tag{1}$$

We observe that there are 18 common key guesses required for the differentials D_1 and D_2. Hence combining the corresponding array of counters T_1 and T_2 we can get $2^{17.13} \times 2^{17.13}/2^{18} = 2^{16.26}$ candidates for 42 bits. Continuing in the same way we observe that $|D_3^K \cap (D_1^K \cup D_2^K)| = 24$, hence we get $2^{16.26} \times 2^{17.13}/2^{24} = 2^{9.39}$ candidates for 48 bits. Using D_4 this can be further reduced, as $|D_4^K \cap (D_1^K \cup D_2^K \cup D_3^K)| = 28$ we expect $2^{9.39} \times 2^{17.13}/2^{28} \approx 2^{-1.5}$ candidates for 50 bits. For the remaining 46 bits we perform an exhaustive search.

Complexity

The complexity of the attack is dominated by the key recovery process. For the partial decryption process we need $2^{17} \times 2^{30} \times \frac{4}{26} \approx 2^{45}$ encryptions,

[3] The key has no influence on the input to the non-linear function in the last round.

hence the complexity of one key recovery attack is 2^{54}. This key recovery is performed for each differential and each 2^6 round key guesses of the initial rounds. Hence the overall complexity of the attack is $2^{54} \times 2^6 \times 4 = 2^{62}$.

We expect in our attack that at least 4 out of 2^{46} pairs follow our differential, which has probablity $\geqslant 2^{-43.65}$, for the correct key. Therefore we get a success rate of

$$1 - \Pr(X < 4) \approx 0.75 \tag{2}$$

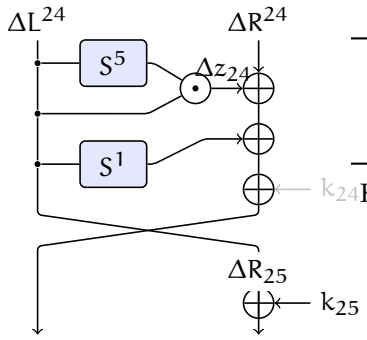
However, in practice this will be much higher as we only use a lower bound on the probability of the differential.

5.2 Key Recovery for 19-round Simeck32

For Simeck32 we also use 4 differentials

$$D_1 : (8000,\ 4011) \xrightarrow{f^{13}} (4000,\ 0000)$$
$$D_2 : (0001,\ 8022) \xrightarrow{f^{13}} (8000,\ 0000)$$
$$D_3 : (0008,\ 0114) \xrightarrow{f^{13}} (0004,\ 0000)$$
$$D_4 : (0010,\ 0228) \xrightarrow{f^{13}} (0008,\ 0000)$$

each having probability $\approx 2^{-28}$ (for the truncated differences see Table 10). We add two rounds at the top of the 13-round differential and identify a set of



ΔR_{25}	∗∗∗0000000000∗000∗∗∗01
$S^5(\Delta R_{25})$	000000000∗000∗∗∗1∗∗∗00
Δz_{24}	∗∗∗0000000∗000∗∗∗∗∗∗0∗
ΔR_{24}	1∗1000000000000000∗000

Key filtering:

1. Find bits s.t. $\Delta z_{24} = *$ and $\Delta R_{24} \neq *$.
2. Guess corresponding bits in k_{25}.
3. Check $\Delta z_{24} = \Delta R_{24} \oplus S^1(R_{25}) \oplus \Delta L_{25}$.

Figure 2: Outline of the process of key guessing and filtering for a single round.

2^{30} pairs of plain-texts each satisfying the specific difference $(\Delta L^2, \Delta R^2)$ after the first two round encryption. Identifying a set of plaintext pairs requires to guess 6 key bits.

Filtering

We can filter some wrong pairs by unrolling the last round and verifying the truncated difference (with 18 known bits) at the beginning of the last round. This will leave us with $2^{30-18} = 2^{12}$ pairs.

Key guessing

We counted that 22 round key bits are necessary to guess for verifying the difference at the end of round 14. The required key bits D_1^K for D_1 are

$$\begin{aligned} K^{16} &= \{3, 9\} \\ K^{17} &= \{2, 3, 4, 8, 9, 10, 14\} \\ K^{18} &= \{1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 13, 14, 15\} \end{aligned}$$

We use the same method as described for Simeck48 during this phase. Out of the filtered pairs we expect to get at least 4 correct pairs (those follow the 13-round differential). Hence the number of candidates for 22 key bits are $\approx 2^{9.1}$. The number of common key bits amongst the differentials is given by

$$\begin{aligned} D_1^K \cap D_2^K &= 14 \\ D_3^K \cap (D_1^K \cup D_2^K) &= 16 \\ D_4^K \cap (D_1^K \cup D_2^K \cup D_3^K) &= 20 \end{aligned}$$

and we expect to 1 key candidate for 38 bits. For the remaining 26 bits of the last four round keys we perform exhaustive search.

Complexity

The complexity of the partial decryption (for the last 4 rounds) is $2^{12} \times 2^{22} \times \frac{4}{19} \approx 2^{32}$ which is the dominating part of the complexity. Since we perform the key recovery for each differential and for each 6-bit round key guesses of the first two rounds the overall complexity of the attack is $2^{32+8} = 2^{40}$.

5.3 Key Recovery for 33-round Simeck64

We use the following 4 differentials for Simeck64

$$D_1 : (0,\ 04400000) \xrightarrow{f^{26}} (08800000,\ 00400000)$$
$$D_2 : (0,\ 44000000) \xrightarrow{f^{26}} (88000000,\ 04000000)$$
$$D_3 : (0,\ 40000004) \xrightarrow{f^{26}} (80000008,\ 40000000)$$
$$D_4 : (0,\ 00000044) \xrightarrow{f^{26}} (00000088,\ 00000004)$$

each having probability $\approx 2^{-60}$ (for the truncated differences see Table 11). We add two rounds at the top of the 26 round differential and identify a set of 2^{62} pairs of plain-texts by guessing 4 round key bits from the first two rounds.

Filtering wrong pairs

We add 5 round truncated difference at the end of the 26 round differential. The last round may be unrolled to verify the difference at the beginning of the last round. This helps to filter some wrong pairs using the known bits of the truncated difference and after filtering we are left with $2^{62-30} = 2^{32}$ pairs of plaintext out of which we expect 2^2 correct pairs (those followed 26 round differential).

Key guessing

In this phase we guess the necessary key bits from the last four rounds to verify the difference at the beginning of round 28. We counted that 76 key bits are necessary to guess for verifying $(\Delta L^{28}, \Delta R^{28})$. The required key bits D_1^K for D_1 are

$$K^{29} = \{0, 18, 22, 28\}$$
$$K^{30} = \{0, 1, 5, 13, 17, 18, 19, 21, 22, 23, 27, 28, 29, 31\}$$
$$K^{31} = \{0, 1, 2, 4-6, 8, 10, 12-14, 16-24, 26-31\}$$
$$K^{32} = \{0-31\}$$

Out of the filtered pairs we expect to get at least 4 correct pairs (those that follow the 26-round differential). Hence the number of candidates for 76 key

bits are $\approx 2^{63.12}$. The number of common key bits amongst the differentials is given by

$$\begin{aligned} D_1^K \cap D_2^K &= 66 \\ D_3^K \cap (D_1^K \cup D_2^K) &= 70 \\ D_4^K \cap (D_1^K \cup D_2^K \cup D_3^K) &= 64 \end{aligned}$$

By combining all the four differentials we expect to get 2^{52} key candidates for 104 bits. For the remaining 24 bits of the last four round keys we perform exhaustive search.

Complexity

The complexity of the partial decryption (for last 4 rounds) is $2^{32} \times 2^{76} \times \frac{5}{33} \approx 2^{105}$ which is the dominating part of the complexity. Since we perform the key recovery for each differential and for each 6-bit round key guesses of the first two rounds the overall complexity of the attack is $2^{105+10} = 2^{115}$.

Table 6: Comparison of the attacks on Simeck.

Cipher	Rounds	Time	Data	Memory	Type
Simeck32/64	20/32	$2^{62.6}$	2^{32}	2^{56}	Imp. Differential [13]
Simeck32/64	22/32	$2^{57.9}$	2^{32}	–	Diff.(dynamic key-guessing) [8]
Simeck32/64	18/32	$2^{63.5}$	2^{31}	–	Linear [3]
Simeck32/64	19/32	$\mathbf{2^{40}}$	$\mathbf{2^{31}}$	$\mathbf{2^{31}}$	Differential (Section 5.2)
Simeck48/96	24/36	$2^{94.7}$	2^{48}	2^{74}	Imp. Differential [13]
Simeck48/96	28/36	$2^{68.3}$	2^{46}	–	Diff.(dynamic key-guessing) [8]
Simeck48/96	24/36	2^{94}	2^{45}	–	Linear [3]
Simeck48/96	26/36	$\mathbf{2^{62}}$	$\mathbf{2^{47}}$	$\mathbf{2^{47}}$	Differential (Section 5.1)
Simeck64/128	25/44	$2^{126.6}$	2^{64}	2^{79}	Imp. Differential [13]
Simeck64/128	34/44	$2^{116.3}$	2^{63}	–	Diff.(dynamic key-guessing) [8]
Simeck64/128	35/44	$2^{116.3}$	2^{63}	–	Diff.(dynamic key-guessing) [8]
Simeck64/128	27/44	$2^{120.5}$	2^{61}	–	Linear [3]
Simeck64/128	33/44	$\mathbf{2^{115}}$	$\mathbf{2^{63}}$	$\mathbf{2^{63}}$	Differential (Section 5.3)

6 Conclusion and Future Work

We gave a brief overview of the Simeck and Simon block cipher and their resistance against differential and linear cryptanalysis. From our comparison we can see that statistical attacks can cover a significant larger number of rounds for Simeck48 and Simeck64. Our key recovery attacks still have a significant margin compared to generic attacks (see Table 6) in regard to time

complexity, therefore additional rounds can be covered using the dynamic key-guessing approach at the costs of a higher complexity.

This also shows that the impact of small design changes in Simon-like block ciphers can be hard to estimate and requires a dedicated analysis, as the underlying design strategy is still not well understood. Especially for variants with a larger block size it is difficult to find lower bounds or estimate the effect of differentials. An open question is whether better differentials exist for both Simon and Simeck which give a surprisingly higher probability as in the case of our differential for Simeck32. This effect could be more significant for larger word sizes and lead to improved attacks.

In this sense Simeck also has an unexpected advantage over Simon and Speck, as the analysis is simpler and requires less computational effort with our approach. This is a property that is especially important in the light of not having cryptanalytic design documentation, nor design rationales for the constants regarding security available by the designers of Simon and Speck.

For both Simon32 and Simeck32 reduced to 13 rounds we observed that for some keys a surprisingly large number of valid pairs can be found. This gives an interesting open problem in classifying the keys which give a significant higher probability for a given differential.

References

[1] Farzaneh Abed, Eik List, Stefan Lucks, and Jakob Wenzel. "Differential Cryptanalysis of Round-Reduced SIMON and SPECK." In: *Fast Software Encryption, FSE 2014*. Ed. by Carlos Cid and Christian Rechberger. Vol. 8540. Lecture Notes in Computer Science. Springer, 2015, pp. 525–545. ISBN: 978-3-662-46705-3.

[2] Javad Alizadeh, Hoda AlKhzaimi, Mohammad Reza Aref, Nasour Bagheri, Praveen Gauravaram, Abhishek Kumar, Martin M. Lauridsen, and Somitra Kumar Sanadhya. "Cryptanalysis of SIMON Variants with Connections." In: *Radio Frequency Identification: Security and Privacy Issues, RFIDSec 2014*. Ed. by Nitesh Saxena and Ahmad-Reza Sadeghi. Vol. 8651. Lecture Notes in Computer Science. Springer, 2014, pp. 90–107. ISBN: 978-3-319-13065-1.

[3] Nasour Bagheri. "Linear Cryptanalysis of Reduced-Round SIMECK Variants." In: *Progress in Cryptology - INDOCRYPT 2015*. 2015, pp. 140–152.

[4] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. *The SIMON and SPECK Families of Lightweight Block Ciphers*. Cryptology ePrint Archive, Report 2013/404. http://eprint.iacr.org/. 2013.

[5] Alex Biryukov, Arnab Roy, and Vesselin Velichkov. "Differential Analysis of Block Ciphers SIMON and SPECK." In: *Fast Software Encryption, FSE 2014*. Ed. by Carlos Cid and Christian Rechberger. Vol. 8540. Lecture Notes in Computer Science. Springer, 2015, pp. 546–570. ISBN: 978-3-662-46705-3.

[6] Stefan Kölbl. *CryptoSMT: An easy to use tool for cryptanalysis of symmetric primitives*. https://github.com/kste/cryptosmt. 2015.

[7] Stefan Kölbl, Gregor Leander, and Tyge Tiessen. "Observations on the SIMON Block Cipher Family." In: *Advances in Cryptology - CRYPTO 2015*. 2015, pp. 161–185.

[8] Kexin Qiao, Lei Hu, and Siwei Sun. *Differential Security Evaluation of Simeck with Dynamic Key-guessing Techniques*. Cryptology ePrint Archive, Report 2015/902. http://eprint.iacr.org/. 2015.

[9] Siwei Sun, Lei Hu, Meiqin Wang, Peng Wang, Kexin Qiao, Xiaoshuang Ma, Danping Shi, Ling Song, and Kai Fu. *Constructing Mixed-integer Programming Models whose Feasible Region is Exactly the Set of All Valid Differential Characteristics of SIMON*. Cryptology ePrint Archive, Report 2015/122. http://eprint.iacr.org/. 2015.

[10] Siwei Sun, Lei Hu, Peng Wang, Kexin Qiao, Xiaoshuang Ma, and Ling Song. "Automatic Security Evaluation and (Related-key) Differential Characteristic Search: Application to SIMON, PRESENT, LBlock, DES(L) and Other Bit-Oriented Block Ciphers." In: *Advances in Cryptology - ASIACRYPT 2014*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science. Springer, 2014, pp. 158–178. ISBN: 978-3-662-45610-1.

[11] Ning Wang, Xiaoyun Wang, Keting Jia, and Jingyuan Zhao. *Differential Attacks on Reduced SIMON Versions with Dynamic Key-guessing Techniques*. Cryptology ePrint Archive, Report 2014/448. http://eprint.iacr.org/. 2014.

[12] Qingju Wang, Zhiqiang Liu, Kerem Varici, Yu Sasaki, Vincent Rijmen, and Yosuke Todo. "Cryptanalysis of Reduced-Round SIMON32 and SIMON48." In: *Progress in Cryptology - INDOCRYPT 2014*. Ed. by Willi Meier and Debdeep Mukhopadhyay. Vol. 8885. Lecture Notes in Computer Science. Springer, 2014, pp. 143–160. ISBN: 978-3-319-13038-5.

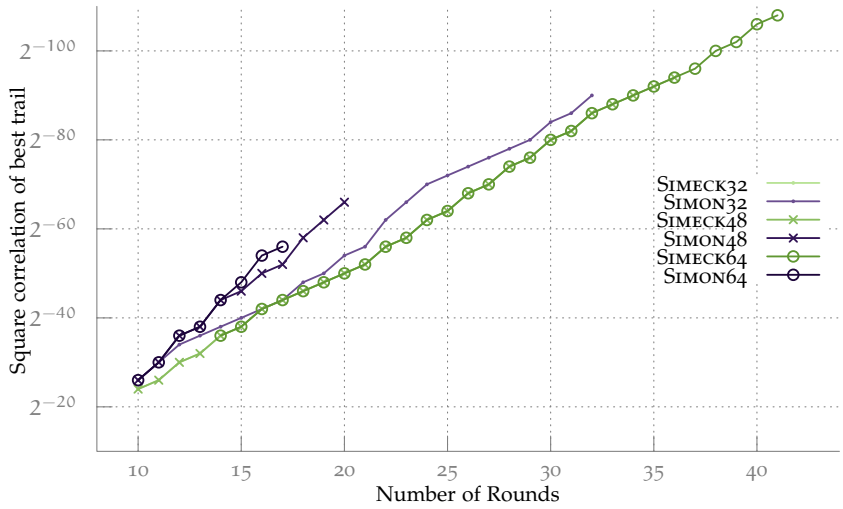


Figure 1: Bounds for the best linear trails for variants of Simon and Simeck. For the different variants of Simeck the bounds are the same.

[13] Gangqiang Yang, Bo Zhu, Valentin Suder, Mark D. Aagaard, and Guang Gong. "The Simeck Family of Lightweight Block Ciphers." In: *Cryptographic Hardware and Embedded Systems - CHES 2015*. to appear. Springer, 2015.

A Bounds for Linear trails

Table 7: Classification of all the 40 rotation invariant 13-round differentials for Simeck32.

Type 1			
$(0,22) \xrightarrow{f^{13}} (2a,1)$	$(4,8a8) \xrightarrow{f^{13}} (88,0)$	$(4,8e8) \xrightarrow{f^{13}} (88,)$	$(0,11) \xrightarrow{f^{13}} (1d,8)$
$(0,11) \xrightarrow{f^{13}} (115,8)$	$(0,88) \xrightarrow{f^{13}} (8e8,4)$	$(4,a8) \xrightarrow{f^{13}} (88,0)$	$(1,3a) \xrightarrow{f^{13}} (22,0)$
Type 2			
$(4,8a) \xrightarrow{f^{13}} (aa,4)$	$(4,8a) \xrightarrow{f^{13}} (ae,4)$	$(1,a8) \xrightarrow{f^{13}} (228,1)$	$(4,aa) \xrightarrow{f^{13}} (a,4)$
$(4,8e) \xrightarrow{f^{13}} (aa,4)$	$(4,2e) \xrightarrow{f^{13}} (a,4)$	$(4,2e) \xrightarrow{f^{13}} (e,4)$	$(2,57) \xrightarrow{f^{13}} (5,2)$
$(2,5) \xrightarrow{f^{13}} (55,2)$	$(4,8e) \xrightarrow{f^{13}} (2a,4)$	$(1,2a8) \xrightarrow{f^{13}} (228,1)$	$(2,7) \xrightarrow{f^{13}} (55,2)$
$(4,aa) \xrightarrow{f^{13}} (8e,4)$	$(4,ae) \xrightarrow{f^{13}} (e,4)$	$(4,8a) \xrightarrow{f^{13}} (2e,4)$	$(2,15) \xrightarrow{f^{13}} (5,2)$
$(2,7) \xrightarrow{f^{13}} (17,2)$	$(4,e) \xrightarrow{f^{13}} (ae,4)$	$(4,ae) \xrightarrow{f^{13}} (8e,4)$	$(4,8a) \xrightarrow{f^{13}} (2a,4)$
$(4,e) \xrightarrow{f^{13}} (2a,4)$	$(4,a) \xrightarrow{f^{13}} (2a,4)$	$(4,2e) \xrightarrow{f^{13}} (8a,4)$	$(4,2a) \xrightarrow{f^{13}} (8e,4)$
$(4,a) \xrightarrow{f^{13}} (ae,4)$	$(4,8e) \xrightarrow{f^{13}} (ae,4)$	$(1,28) \xrightarrow{f^{13}} (b8,1)$	$(4,8e) \xrightarrow{f^{13}} (2e,4)$
$(1,b8) \xrightarrow{f^{13}} (238,1)$	$(4,ae) \xrightarrow{f^{13}} (8a,4)$	$(2,15) \xrightarrow{f^{13}} (7,2)$	$(1,2a8) \xrightarrow{f^{13}} (38,1)$

Table 8: Distribution of the trails for the different type of differentials in 13-round Simeck32.

$\log_2 \Pr(Q)$	Type 1	Type 2	Type 3
−32	1	1	0
−33	0	0	0
−34	9	7	0
−35	6	5	0
−36	38	24	8
−37	44	28	2
−38	124	71	87
−39	166	96	79
−40	367	210	560
−41	521	308	868
−42	1014	625	2911
−43	1566	1002	5170
−44	2629	1752	12485
−45	4232	2975	22007
−46	6448	5101	43969
−47	9620	8234	75212
−48	13952	14439	133341
−49	19425	24653	220359
$\sum$	$2^{-27.88}$	$2^{-28.43}$	$2^{-27.29}$

Table 9: Number of trails and time to find them for the SIMECK48 differential $(400000, e00000) \xrightarrow{f^{20}} (400000, 200000)$.

$\log_2$ Pr(Q)	#Trails	Pr(Differential)	T
−50	1	−50.0	3.72s
−51	0	−50.0	6.9s
−52	12	−48.0	19.78s
−53	6	−47.7520724866	31.77s
−54	80	−46.7145977811	42.62s
−55	68	−46.4301443917	55.68s
−56	413	−45.804012702	77.58s
−57	484	−45.5334136623	104.69s
−58	1791	−45.1367816524	180.02s
−59	2702	−44.8963843436	265.5s
−60	7225	−44.6271009401	528.39s
−61	12496	−44.4289288164	1068.95s
−62	28597	−44.2312406041	2603.59s
−63	52104	−44.0720542548	6146.77s
−64	111379	−43.9193398907	19276.9s
−65	207544	−43.7902765446	41938.08s
−66	238939	−43.7209043818	70720.98s
−67	228530	−43.6888725691	96657.81s
−68	229018	−43.6730860168	123706.38s
−69	276314	−43.6636455186	160688.8s
−70	271192	−43.6590352669	197354.41s
−71	269239	−43.6567522016	232641.34s
−72	267563	−43.6556191172	271083.28s
−73	266716	−43.6550547005	308072.68s
−74	227971	−43.6548135551	336027.17s

Table 10: Truncated differential for SIMECK32 obtained by extending $(8000, 4011) \xrightarrow{f^{13}} (4000, 0)$ in both directions until all bits are unknown.

Round	ΔL	ΔR	*	*
−4	***0**********	**********	15	16
−3	**000***0****1**	***0**********	11	15
−2	0*0000*000***01*	**000***0****1**	6	11
−1	0100000000010001	0*0000*000***01*	0	6
0	1000000000000000	0100000000010001	0	0
	13 rounds			
13	0100000000000000	0000000000000000	0	0
14	1*0000000000*000	0100000000000000	2	0
15	**00000*000**001	1*0000000000*000	5	2
16	***000**00***01*	**00000*000**001	9	5
17	***00***0******	***000**00***01*	13	9
18	***0**********	***00***0******	15	13
19	**********	***0**********	16	15

Table 11: Truncated differential for SIMECK64 obtained by extending $(0, 4400000) \xrightarrow{f^{26}} (8800000, 400000)$ in both directions until all bits are unknown.

Round	ΔL	ΔR	*	*
−8	**********0**********	**********	31	32
−7	**********0**00***0**********	**********0**********	28	31
−6	**********00*000**00***0********	**********0**00***0**********	24	28
−5	**********0000000*000**00***0***	**********00*000**00***0********	19	24
−4	*0****1***0000000000000*000**00**	**********0000000*000**00***0***	13	19
−3	*00***01**00000000000000000*000*	*0****1***000000000000*000**00**	8	13
−2	*000**001*0000000000000000000000	*00***01**0000000000000000000*000*	4	8
−1	00000100010000000000000000000000	*000**001*0000000000000000000000	0	4
0	00000000000000000000000000000000	00000100010000000000000000000000	0	0
		26 rounds		
26	00001000100000000000000000000000	00000000010000000000000000000000	0	0
27	000**001*1000000000000000000000*	00001000100000000000000000000000	4	0
28	00***01***00000000000000000*000**	000**001*1000000000000000000000*	9	4
29	0****1****000000000000*000**00***	00***01***0000000000000000*000**	14	9
30	**********000000*000**00***0****	0****1****00000000000*000**00***	20	14
31	**********0*000**00***0*********	**********000000*000**00***0****	25	20
32	***********00***0********------	**********0*000**00***0*********	29	25
33	***********0********------	***********00***0********------	31	29
34	------	***********0********------	32	31

Haraka v2 – Efficient Short-Input Hashing for Post-Quantum Applications

Publication Information

Stefan Kölbl, Martin M. Lauridsen, Florian Mendel, and Christian Rechberger. *Haraka v2 - Efficient Short-Input Hashing for Post-Quantum Applications*. Cryptology ePrint Archive, Report 2016/098. http://eprint.iacr.org/2016/098. 2016

Contribution

- Main contributions are the cryptanalysis of Haraka and the SPHINCS benchmarks.

Remarks

This publication has been slightly edited to fit the format.

Haraka v2 – Efficient Short-Input Hashing for Post-Quantum Applications

Stefan Kölbl[1], Martin M. Lauridsen[1], Florian Mendel[2], and Christian Rechberger[1,2]

[1] DTU Compute, Technical University of Denmark, Denmark
[2] InfoSec Global Ltd., Switzerland
[3] IAIK, Graz University of Technology, Austria
stek@dtu.dk
martin.lauridsen@infosecglobal.com
{christian.rechberger,florian.mendel}@iaik.tugraz.at

Abstract. Recently, many efficient cryptographic hash function design strategies have been explored, not least because of the SHA-3 competition. These designs are, almost exclusively, geared towards high performance on long inputs. However, various applications exist where the *performance on short (fixed length) inputs matters more*. Such hash functions are the bottleneck in hash-based signature schemes like SPHINCS or XMSS, which is currently under standardization. Secure functions specifically designed for such applications are scarce. We attend to this gap by proposing two short-input hash functions (or rather simply compression functions). By utilizing AES instructions on modern CPUs, our proposals are the fastest on such platforms, reaching throughputs *below one cycle per hashed byte* even for short inputs, while still having a *very low latency* of less than 60 cycles.

Under the hood, this results comes with several innovations. First, we study whether the number of rounds for our hash functions can be reduced, if only second-preimage resistance (and not collision resistance) is required. The conclusions is: only a little. Second, since their inception, AES-like designs allow for supportive security arguments by means of counting and bounding the number of active S-boxes. However, this ignores powerful attack vectors using truncated differentials, including the powerful rebound attacks. We develop a general tool-based method to include arguments against attack vectors using truncated differentials.

Keywords: Cryptographic hash functions, second-preimage resistance, AES-NI, hash-based signatures, post-quantum

1 Introduction

Cryptographic hash functions are commonly constructed with collision resistance in mind. Consider e.g. the SHA-3 competition, which involved a large part of the research community, where collision resistance was one of the main requirements. Sometimes, cryptographic functions are designed with collision resistance as the main or only requirement, see e.g. VSH [15]. This is in contrast to a sizable and growing set of applications, that utilize cryptographic hashing, but explicitly *do not require collision resistance*. Consider as an example the proof for the HMAC construction, which initially required collision resistance from its hash function [4], but in later versions the collision resistance requirement was dropped in favor of milder requirements [3]. Universal one-way hash functions (UOWHF) [5] are, in principle, candidate functions, but they will not suffice for many applications.

Another example, which brings us to the main use-case of this paper, are hash-based signature schemes originally introduced by Lamport [36]. Recent schemes include XMSS [13], which is currently submitted as a draft standard to the IETF and which features short signatures sizes, and the state-less scheme SPHINCS [9]. A recent version of the former, XMSS-T [25], attains additional security against multi-target preimage attacks on the underlying hash function. Arguably, such designs are the most mature candidates for signature schemes offering post-quantum security, i.e. they are believed to be secure in the presence of hypothetical quantum computers, as their security reduces *solely* to the security properties of the hash function(s) used, thus relying on minimal assumptions.

The hash-based signature schemes mentioned require many calls to a hash function, but only process short inputs. For instance in SPHINCS-256, about 500000 calls to two hash functions are needed to reach a post-quantum security level of 128 bits. One of those functions (denoted H) compresses a 512-bit string to a 256-bit string and is used in a Merkle-tree construction, while the other (denoted F) maps a 256-bit string to a 256-bit string.

The applications share the absence of collision resistance from the requirements imposed on the underlying hash function(s), and further they process only short inputs[4]. However, nearly all cryptographic hash functions are geared towards high performance on long messages and, as we will show, perform rather poorly on short inputs.

[4]For HMAC, one of the two calls to the hash function used is always for a short input.

1.1 Contributions

Motivated by the applications described above, we explicitly consider preimage- and second-preimage resistance as the sole security goals for cryptographic hash functions, particularly dropping collision resistance, and furthermore target high performance on short (fixed length) inputs. We limit ourselves to one particular design strategy, which is fairly well understood and scalable: AES-like designs. This enables both strong security arguments, while also allowing excellent performance on widespread platforms offering AES-specific instructions, such as modern Intel and AMD CPUs, as well as the ARMv8 architecture.

Concretely we propose Haraka v2, two secure (in the above sense) short-input hash functions achieving a performance better than 1 cycle per byte (cpb) and a latency of only 60 cycles, on various Intel architectures. As we show in Section 5.3, competitive designs are somewhat slower than that. Our proposals share strong similarities with the permutation AESQ that is used in the CAESAR candidate PAEQ [11]. We perform benchmarks of the SPHINCS-256 hash-based signature, using Haraka v2 as the underlying hash functions, and show performance speed-ups of $\times 1.50$ to $\times 2.86$. As hash-based signature schemes such as SPHINCS are already practical, and in the case of XMSS in the process of standardization, this shows that Haraka v2 can significantly contribute to speeding up such schemes.

On the theoretical side, our proposal also carries with it several contributions. Firstly, we study if the number of rounds for Haraka v2 can be reduced if only second-preimage resistance, and *not collision resistance*, is required. The conclusion is that only one round (5 rounds instead of 6) can be dropped. Secondly, and as a point we like to elaborate at this point already, we describe new ways to bound the applicability of attacks. Traditionally, resistance against differential attacks (which are important for collision- and second-preimage attacks) of key-less constructions, such as cryptographic hash functions, is almost solely based on arguments that are also found for keyed constructions such as block ciphers. Common approaches include (1) using a bound on the best differential trail and comparing it with the available degrees of freedom, or (2) assuming a number of rounds *controlled* by the attacker, and use a bound on the best differential trail for the *uncontrolled* rounds as a security margin. Such arguments have been used for various SHA-3 candidates like Grøstl [20], ECHO [6], Luffa [16], and the more recent hash function PHOTON [23]. One problem of these approaches is that they do not consider truncated differentials, and as such do not cover rebound attacks.

Arguments against rebound attacks are of course still possible and can be found in the literature, also for the aforementioned designs. Often this

involves designing concrete rebound attacks along with arguments for why improving them is unlikely. Alternatively, designers make assumptions akin to (2) about the *controlled* rounds that are simply "for free", and then focus on bounding the effect of the *uncontrolled* rounds. Perhaps the most notable arguments in that direction are for the design of SPN-Hash [14], which uses approach (2), but provides bounds for the *uncontrolled* rounds using differentials and not solely single trails. However, the *controlled* rounds are still treated as a black box.

To improve the situation, we propose a way to model an idealized attacker who has capabilities which resemble cryptanalytic techniques such as the rebound attack. We take into account how the complexity of an attack can be reduced in the controlled rounds, like in the inbound phase of the rebound attack, by using the available degrees of freedom to fulfill conditions in a truncated differential. This allows us better security arguments by not having to treat parts of the hash function as a black box, and we can take into account also the fact that there are less degrees of freedom available in a second-preimage attack. Overall, this gives us a better understanding of the required number of rounds for Haraka v2 to resist these types of attacks.

Finally, we remark at this point that both implementations of our proposals, including test vectors, the SPHINCS code for benchmarking and the code used to generate the MILP models for the security analysis of Haraka v2, are publicly available[5].

1.2 Related Work

Several proposals have been submitted to the SHA-3 competition that aim to take advantage of AES instructions in modern CPUs. Among them are Grøstl [20], ECHO [6], Fugue [24], LANE [28]. Many of them are geared towards performance on long messages, and often show severe performance degradation for short messages. The CAESAR competition for authenticated encryption schemes saw many proposals, including AEGIS [49], PAEQ [11] and Tiaoxin [40], based on utilizing AES instructions. Recently, two designs for permutations based on the AES round function have been proposed. Jean and Nikolic [31] studied AES-based designs for MACs and authenticated encryption, however not for hashing applications. Gueron and Mouha [22] propose Simpira v2, a family of permutations based on Feistel networks.

[5]See supplementary material

1.3 Recent Developments in Short-Input Hashing

Haraka v1 was originally presented to a larger group of cryptographers in November 2015 [41], with the explicit goal of providing fast hashing on short inputs, the main application being speeding up hash-based signature schemes. Simpira, mentioned above, started circulating a few months thereafter, with one of the three main applications mentioned also being hash-based signature schemes [22, Section 7]. Haraka v1 was broken by Jean [30] (see also Section 4.2), and in this paper Haraka v2 is presented, which differs from Haraka v1 in the choice of round constants. Simpira (the former version) was broken in two different ways [18, 43] and Simpira v2 addresses the identified problems. Concrete performance numbers for Simpira v2 in modern hash-based signature schemes are not available yet, but our benchmarks in Section 5.3 suggest that Simpira v2 is slower.

Recently, KangarooTwelve, a variant of Keccak with a reduced number of rounds, was proposed, aimed at improved hashing speed. However, its performance is still geared towards long inputs. Furthermore, improvements on SHA-256 implementations are being discussed in the community. In Section 5.3 we discuss briefly recent performance figures on Skylake for SHA-256 as well as comparison with KangarooTwelve.

Secure short-input keyed hash functions also found applications in protecting against *hash flooding* denial of service attacks. This has been addressed with the SipHash [1] family, but the security requirements are much lower for this setting.

2 Specification of Haraka v2

Haraka v2 exists in two variants denoted Haraka-512 v2 and Haraka-256 v2 with signatures

$$\begin{aligned} &\text{Haraka-512 v2} : \mathbb{F}_2^{512} \to \mathbb{F}_2^{256} \quad \text{and} \\ &\text{Haraka-256 v2} : \mathbb{F}_2^{256} \to \mathbb{F}_2^{256}. \end{aligned} \tag{1}$$

For both variants, we claim 256 bits of security (respectively 128 bits in the presence of quantum computers) against (second)-preimage resistance, but we make *no further claims* about other non-random properties.

The main components are two permutations denoted π_{512} and π_{256} on 512 bits and 256 bits, respectively. Both Haraka-512 v2 and Haraka-256 v2 employ the well-known Davies-Meyer (DM) construction using a permutation with

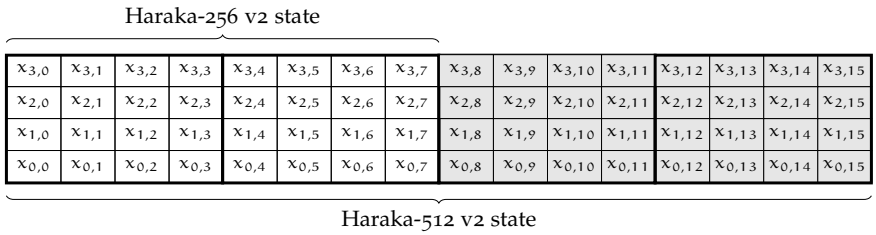


Figure 1: State for Haraka-512 v2 and Haraka-256 v2 (not including the shaded area). The box $x_{i,j}$ denotes the ith byte in the jth column of the state.

a feed-forward (applying the XOR operation) of the input. As such, they are defined as

$$\begin{aligned} &\text{Haraka-512 v2}(x) = \mathsf{trunc}(\pi_{512}(x) \oplus x) \quad \text{and} \\ &\text{Haraka-256 v2}(x) = \pi_{256}(x) \oplus x, \end{aligned} \tag{2}$$

where $\mathsf{trunc} : \mathbb{F}_2^{512} \to \mathbb{F}_2^{256}$ is a particular truncation function (described below).

2.1 Specification of π_{512} and π_{256}

In the following, we give our specification of the permutations used in Haraka v2. In Section 3, we give our security analysis of the constructions and, based on this, motivate our design choices in Section 4.4.

The constructions of π_{512} and π_{256} are *iterated*, thus applying a *round function* several times to obtain the full permutation. The permutations π_{512} and π_{256} operate on *states* which have the same size as respective inputs. Due to the similarity of the permutations, much of their description is common to both. When we talk about a *block*, we refer to a 16-byte string consisting of four columns denoted $x_{4i} \| \cdots \| x_{4i+3}$ for $i = 0, \ldots, b-1$. In general, we let b denote the number of 128-bit blocks of the state, so for π_{512} we have $b = 4$ while for π_{256} we have $b = 2$. The state arrangement is given in Figure 1.

Denote the total number of rounds by T and denote by R_t the round with index $t = 0, \ldots, T-1$. The state *before* applying R_t is denoted S^t, and thus S^0 is the initial state. As both π_{512} and π_{256} use the AES round function, states are arranged in matrices of bytes, and we use subscripts to denote the column index, starting from column zero being the leftmost one. The state size is $4 \times 4b$ bytes, so 4×16 for π_{512} and 4×8 for π_{256}. When a stream of bytes is loaded into the state, the order is column first, such that the first byte of the input stream is in the first row of the first column, while the last byte of the stream is in the last row of the last column.

Let **aes** denote the parallel application of m AES rounds to each of the b blocks of the state. As such, for $t = 0, \ldots, T-1$, the round function for π_{512} is $R_t = \mathbf{mix}_{512} \circ \mathbf{aes}$ while for π_{256} it is $R_t = \mathbf{mix}_{256} \circ \mathbf{aes}$. Thus, in both cases, a single round consists of m rounds of the AES applied to each block of the state, followed by a linear mixing function. Round constants are injected via the **aes** operations (see below). The number of rounds is $T = 5$ while using $m = 2$ AES rounds for both Haraka-512 v2 and Haraka-256 v2 (totaling 10 AES rounds).

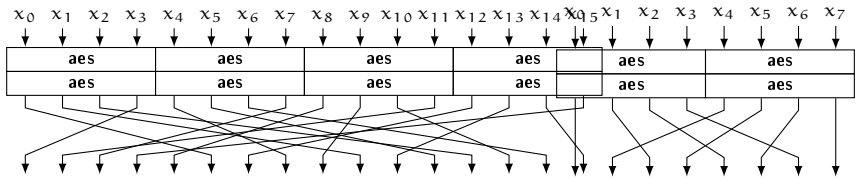
The main difference π_{512} and π_{256} is the linear mixing used. In both cases, the mixing itself is comprised of simply permuting the state columns. For π_{512}, the sixteen columns of the state are permuted such that each output block contains precisely one column from each of the $b = 4$ input blocks. For π_{256} on the other hand we have $b = 2$ so we obtain the most even distribution of the columns by mapping two columns from each of the $b = 2$ input blocks to each of the $b = 2$ output blocks. Letting $x_0 \| \cdots \| x_{15}$ denote the columns for a state of π_{512}, the columns are permuted by $\mathbf{mix}_{512}$ as

$$x_0 \| \cdots \| x_{15} \mapsto x_3 \| x_{11} \| x_7 \| x_{15} \| x_8 \| x_0 \| x_{12} \| x_4 \| x_9 \| x_1 \| x_{13} \| x_5 \| x_2 \| x_{10} \| x_6 \| x_{14}. \tag{3}$$

Likewise for π_{256} the eight columns denoted $x_0 \| \cdots \| x_7$ are permuted by $\mathbf{mix}_{256}$ as

$$x_0 \| \cdots \| x_7 \mapsto x_0 \| x_4 \| x_1 \| x_5 \| x_2 \| x_6 \| x_3 \| x_7. \tag{4}$$

The round functions for both permutations are depicted in Figure 2.



(a) For π_{512} (b) For π_{256}

Figure 2: Depictions of round functions R_t for π_{512} (a) and for π_{256} (b). Each x_i denotes a column of 4 bytes of the state.

Round Constants

For each AES call, we use different round constants via the round key addition. The constants are derived using a similar approach as in the CAESAR

candidate Prøst [32]. Let p_i be the least significant bit of the ith digit after the decimal point of π, then the round constants are defined as

$$RC_j = p_{128j+128} \| \dots \| p_{128j+2} \| p_{128j+1} \qquad \forall j = 0 \dots 39. \tag{5}$$

The AES layer $\mathbf{aes}_i$ uses round constants $(RC_{4i}, RC_{4i+1}, RC_{4i+2}, RC_{4i+3})$ in the case of π_{512}, respectively (RC_{2i}, RC_{2i+1}) for π_{256}. The constants are also given in Section A. The use of π is an application of a nothing-up-my-sleeve number; another choice of a known constant would be an equally qualified.

Truncation Function

Let $x \in \mathbb{F}_2^{512}$. Then $\mathsf{trunc}(x)$, which is used in Haraka v2, is obtained as concatenating two columns from each block: The least significant two from the first two blocks, and the two most significant columns from the last two blocks. As such

$$\mathsf{trunc}(x_0 \| \cdots \| x_{15}) = x_2 \| x_3 \| x_6 \| x_7 \| x_8 \| x_9 \| x_{12} \| x_{13}. \tag{6}$$

3 Security Requirements

The three most commonly defined security requirements for a cryptographic hash function H are

- **Preimage resistance:** Given an output y it should be computationally infeasible to find any input x such that $y = H(x)$,
- **Second-preimage resistance:** Given $x, y = H(x)$ it should be computationally infeasible to find any $x' \neq x$ such that $y = H(x')$, and
- **Collision resistance:** Finding two distinct inputs x, x' such that $H(x) = H(x')$ should be computationally infeasible.

Generic attacks, which can find a (second-)preimage with a complexity of 2^n and collisions with a complexity of $2^{n/2}$, exist for any hash function, where n is the digest size in bits. Quantum computers can improve upon this by using Grover's algorithm [21] to further reduce the complexity of finding a (second-)preimage to $2^{n/2}$. It is also known that this is the optimal bound for quantum computing. Brassard, Høyer and Tapp's method [12] suggests an algorithm finding collisions in $2^{n/3}$ steps, however the actual costs are not lower compared to methods based on classical computers [7].

In the following sections, we discuss common attack vectors which will aid in choosing appropriate parameters for Haraka v2 to achieve the desired security properties. As described, we focus on second-preimage resistance, as the main applications of Haraka v2 do not require collision resistance.

3.1 Preliminaries

Differential cryptanalysis is a powerful tools for evaluating the security of cryptographic hash functions. It is also a very natural attack vector, as both collision and second-preimage resistance require the attacker to efficiently find two distinct inputs yielding the same output.

Definition 1. A *differential trail* Q is a sequence of differences

$$\alpha_0 \xrightarrow{R_0} \alpha_1 \xrightarrow{R_1} \cdots \xrightarrow{R_{T-1}} \alpha_T \tag{1}$$

in the states S^t, for the application of the function on two distinct inputs.

Definition 2. The *differential probability* of a differential trail Q is defined as

$$DP(Q) = Pr(\alpha_0 \to \alpha_1 \to \ldots \to \alpha_T) = \prod_{t=0}^{T-1} Pr(\alpha_t \to \alpha_{t+1}) \tag{2}$$

and gives the probability, taken over random choices of the inputs, that the pair follows the differential trail (i.e. the differences match). The last equality holds if we assume independent rounds.

The AES round function uses the SubBytes, ShiftRows and MixColumns operations (denoted SB, SR and MC for short). For our further analysis we will be interested in how truncated differentials [34] propagate through MixColumns. The branch number of MixColumns is 5, so if an input column to MixColumns contains a active bytes, then the probability of having b active bytes in the corresponding output column, where $a + b \geqslant 5$ and $1 \leqslant a, b \leqslant 4$, can be approximated by $2^{(b-4)8}$.

Differential Trails

One way to estimate $DP(Q)$ for the best trail is to count the minimum number of active S-boxes. As the maximum differential probability for the AES S-box is 2^{-6} this allows to give an upper bound on $DP(Q)$. While the number of active S-boxes gives a good estimate for the costs of an attack in the block cipher setting, this is only partially true for cryptographic hash functions.

Consider a pair of inputs $(x, x \oplus \alpha)$ as input to a non-linear function, like the AES S-box, then $S(x \oplus K) \oplus S(x \oplus \alpha \oplus K) = \beta$ holds only with a certain probability if the key K is unknown. This can be very useful in the block cipher settings, where it gives a bound on the probability of the best differential trail. In the case of hash functions there is no secret key and an attacker has full control over the input. This allows him to choose the pair $(x, x \oplus \alpha)$ such that $S(x) \oplus S(x \oplus \alpha) = \beta$ holds with probability 1. The limit of this approach

is only restricted by the number of free and independent values, referred to as *degrees of freedom*. This means that the probability of a differential trail can be very low and contain many active S-boxes, but if the conditions are easy to fulfill, and the attacker has enough degrees of freedom, an attack can be very efficient.

A popular technique to count the number of active S-boxes for AES-based designs is based on *mixed integer linear programming* (MILP) [39, 47]. The basic idea is to express the restrictions on the trail, given by the round transformations, as linear equations, and generate a optimization problem which can be solved with any MILP optimizer, e.g. Gurobi [27] or CPLEX [26]. We use this technique later to find the minimum number of active S-boxes for Haraka v2, which aids us in an informed choice of parameters.

3.2 Capabilities of an Attacker

One of the main difficulties in the design of hash functions is to estimate the security margin one expects against a powerful attacker. As described, bounding the probability of trails can be useful for block ciphers but are of limited use for hash functions, as there is no secret input. Degrees of freedom can be used, to some extent, to solve many conditions on the trail and lead to surprisingly efficient attacks. This was partially addressed in the design of Fugue [24] and SPN-hash [14]. The former assumes that an attacker can improve the probability of a differential trail by using the degrees of freedom directly, i.e. if one has f degrees of freedom the probability can be improved by 2^f. SPN-hash assumes the attacker can bypass r_2 rounds, estimated based on existing attacks, and the total number of rounds is given by $r = r_1 + r_2$, where r_1 is chosen such that the probability of the best differential is low enough for the required security level. A major drawback of this approach is that they do not resemble the capabilities of an attacker in practice, which can lead to too conservative estimates while also ignoring important attack vectors.

The most powerful collision attacks on AES-based hash functions, such as the rebound attack [38], use truncated differentials combined with a clever use of the degrees of freedom to reduce the attack complexity. Arguing security against this type of attacks is a difficult task, as one has to estimate the limits of an attacker to use the available degrees of freedom in a smart way. In the second-preimage scenario, the attacker has much less control as the actual values of the state are fixed, and the conditions are instead solved by carefully choosing the trails. In the following, we propose a new method to better bound the capabilities of an attacker in practice under reasonable assumptions.

Truncated Differentials

A MILP model to count the number of active S-boxes inherently uses truncated differentials (at the byte level), as it considers active bytes but not the difference values, but it does not cover the costs of their propagation. When an attacker tries to utilize a truncated differential, the transitions through MixColumns are probabilistic and, if not controlled by the attacker, will determine the attack complexity similar to the outbound phase in the rebound attack.

An attacker can always use a (fully active) truncated differential with probability ≈ 1 (as a fully active state will very likely remain fully active after MixColumns), which gives a valid second-preimage if the input difference is equal to the output difference. This happens with a probability of 2^{-256}, hence the security can be at most 256 bits for this attack vector.

Utilizing Degrees of Freedom

The previous approaches still ignore the fact that a powerful attacker can utilize the available degrees of freedom to reduce the attack complexity. To take this into account we assume the attacker is able to use all degrees of freedom in an *optimal way*, i.e. the attacker has an algorithm to solve any condition in constant time, as long as there are enough independent degrees of freedom left.

Without any further restrictions we can not achieve any level of security in this model, as the attacker can always use a truncated differential which is active in all bytes having a probability of 1 and then use the degrees of freedom to guarantee the condition $f(x) \oplus f(x \oplus \alpha) = 0$. In general the state size is at least as big as the output size, hence the attacker will have enough degrees of freedom to solve these conditions.

However, it is very unlikely that an attacker can utilize the degrees of freedom in this way without further restrictions. In the case of AES, already after two rounds we get full diffusion, i.e. every byte of the output depends on all bytes of the input. In general solving a condition like $f(x) \oplus f(x \oplus \alpha) = 0$ then corresponds to solving a system of non-linear equations over $\mathbb{F}_{2^8}$ which is an NP-hard problem.

The model we propose is more restrictive and reflects the capabilities of an attacker in practice. The attacker is still allowed to solve conditions for *free* using the degrees of freedom, but can only do so for q consecutive rounds of the primitive. This means, the attacker chooses a state S^k and then is allowed to solve any conditions for states $S^{k-q}, \ldots, S^{k+q}$ in *constant time*, as long as there are still degrees of freedom available. The remaining conditions which can not be solved make up the security level. We can formulate this as a

MILP problem with the goal of finding the lowest attack complexity over all possible states S^k (for more details and the application to Haraka v2 see Section 4.2).

This model for truncated differential attacks resembles how collision attacks on cryptographic hash functions *actually* work in practice. The attacker can control how the differences propagate over a part of the state and tries to minimize the conditions in the remaining rounds [38, 48]. The currently best known attacks on AES-based hash functions utilize the degrees of freedom for up to three AES rounds to reduce the complexity of an attack [29, 46]. These results can not be carried over directly to our construction, as we compose our state of four individual (respectively two) AES states. Very recent work on AESQ [2] found that 4 AES rounds can be covered in an inbound phase, albeit at a high cost.

Therefore, we use both $q = 2$ for the collision and second-preimage case, allowing our idealized attacker to cover a 4 rounds with the degrees of freedom to have a comfortable security margin.

4 Analysis of Haraka v2

In the following we give the security claims for Haraka v2 and the security analysis which lead to the proposed parameters.

4.1 Security Claims

We claim second-preimage resistance of 256 bits for Haraka v2 against classical computers. As will been seen later in the paper, for only one additional round (a performance penalty of around 20%) we claim 128 bits of collision resistance. We make no claims against near-collisions or other generalizations of this property, nor against distinguishers of the underlying permutation, because such properties do not seem to be needed in applications like hash-based signature schemes [9, 13]. Overall, this leads to a conjectured post-quantum security level of 128 bits against both collision and second-preimage attacks.

Non-randomness that might slightly speed-up second-preimage attacks is not excluded by our models and bounds, but we conjecture this to be negligible. To support our conjecture, consider as an example the slight speed-up of second-preimage attacks [17, 19] on the SHA-3 candidate Hamsi [35] which uses a very strong non-random property of the compression function. No such strong property seems likely to exist for our proposals.

4.2 Second-Preimage Resistance

For an output size of $n = 256$ the best generic attacks have a complexity of 2^{256} respectively 2^{128} on a classical- respectively quantum computer. For iterative hash functions, a generic attack exists which improves upon the naïve brute force approach for finding second preimages [33]. However, this attack requires long messages and is therefore not applicable to our construction.

Differential Second-Preimage Attack for Weak Messages

For finding a second-preimage the attacker can use a differential trail Q leading to a collision, that means $f(x \oplus \alpha) = y$. However, as the values of the state are fixed by the output y, all differentials trails hold with probability 1 or 0. For a random message, the probability that an attacker succeeds is bounded by $DP(Q)$, and if Q does not yield a second-preimage for y, then the attacker must try another trail $Q' \neq Q$ or another message.

Table 1: Lower bound on the number of active S-boxes in a differential trail for the permutations used in Haraka v2, for the permutation when used in DM mode and for trails leading to a collision when used in DM mode. Section C gives the numbers for a wider choice of parameters.

	Permutation	DM-mode	DM-mode (coll.)
Haraka-256 v2	80	80	105
Haraka-512 v2	130	128	134

Counting the number of active S-boxes gives a bound on the maximum value of $DP(Q)$ and can give some insights on the security. We consider both the number of active S-boxes for the permutation itself, as well as when the permutation is used in the DM mode. As some of the output is truncated for Haraka-512 v2, this can potentially reduce the number of active S-boxes and has to be taken into account. For Haraka-512 v2 the best differential trail has a probability of $DP(Q) = 2^{-780}$, while the best trail leading to a collision has probability $DP(Q) = 2^{-804}$ when used in DM mode. Similarly, for Haraka-256 v2, those probabilities are 2^{-480} and 2^{-630}, respectively. For the number of active S-boxes for Haraka-512 v2 and Haraka-256 v2 see Table 1. Note that this corresponds to previous work that studied second-preimage attacks for MD4 [51] and SHA-1 [42].

Truncated Differentials

We can use the approach from Section 3.2 to bound the costs of finding a second-preimage for an idealized attacker in order to determine the number

of rounds for Haraka v2. To find a second-preimage the attacker needs to first find a truncated differential leading to a collision and then determine the trail with the available degrees of freedom. However, as the state is fixed by the initial message the degrees of freedom are limited to the choice of differences for each active byte in the truncated trail.

We denote the input column j to MixColumns (resp. SubBytes) in round t as MC_j^t (resp. SB_j^t) and consider the number of rounds T and the number of AES steps per round m, as variables. We define the cost for an attacker to fulfill the conditions of a truncated differential, starting at state S^k, as

$$C_{\text{trunc}} = \sum_{t=0}^{T\cdot m-1} \sum_{j=0}^{4b-1} C_{\mathsf{MC}_j}^t \tag{1}$$

where the costs in the forward direction are given by decision variables $C_{\mathsf{MC}_j}^t$ satisfying

$$\forall t : k \leqslant t \leqslant T\cdot m, \forall j : 0 \leqslant j < 4b : \qquad C_{\mathsf{MC}_j}^t \geqslant \left(4 - \sum_{i=0}^{3} \mathsf{SB}_{i,j}^t\right)\cdot 8 \tag{2}$$

and in the backwards direction by

$$\forall t : 0 \leqslant t < k, \forall j : 0 \leqslant j < 4b : \qquad C_{\mathsf{MC}_j}^t \geqslant \left(4 - \sum_{i=0}^{3} \mathsf{MC}_{i,j}^t\right)\cdot 8 \tag{3}$$

where $\mathsf{SB}_{i,j}^t$ resp. $\mathsf{MC}_{i,j}^t$ is 1 if the byte is active and 0 otherwise. Note that here $C_{\mathsf{MC}_j}^t$ corresponds to the $\log_2$ complexity for the transitions through MixColumns (resp. the inverse MixColumns) to satisfy the truncated differential trail.

An additional requirement is, that the input and output difference are equal, in order to get a valid second-preimage, that means $\mathsf{trunc}(x \oplus \alpha) = \mathsf{trunc}(\Delta\pi_{512}(x \oplus \alpha))$. The complexity depends on the number of active bytes at the input which are not truncated

$$C_{\text{collision}} = \sum_{j\in\mathcal{I}_c} \sum_{i=0}^{3} \mathsf{SB}_{i,j}^0 \cdot 8 \tag{4}$$

where $\mathcal{I}_c$ is the set of column indices which are not truncated at the output. The optimization goal for the MILP problem is then given by

$$\textbf{minimize:} \quad C_{\text{collision}} + C_{\text{trunc}}. \tag{5}$$

The requirements for Haraka v2 are that each attack in this model costs at least 2^{256} to have a good security margin. We applied this model to explore

how the security level evolves for different choices of T and m. For every parameter set, we use the MILP model to find the lowest attack costs by searching over all possible starting states S^k. The results are given in Table 2. The time to solve the MILP problem increases quickly with the number of rounds and for the standard parameters ($T = 5$, $m = 2$, $q = 2$) it takes around 17 minutes[6] to find the lower bound for an attack for all possible starting points S^k.

Degrees of Freedom. The previous scenario does not yet take into account the capabilities of an attacker utilizing the available degrees of freedom. For the second-preimage scenario the attacker can freely choose the differences in one of the states S^k to reduce the costs of the attack for q rounds in both directions

$$\mathcal{D} = \sum_{j=0}^{4b-1} \sum_{i=0}^{3} S^k \cdot 8. \tag{6}$$

The costs for an attack are then given by the number of conditions which can be reduced in the controlled rounds $\mathcal{R} = \{r \mid k-q \leqslant r < k+q \wedge 0 \leqslant r < T \cdot m\}$ by using degrees of freedom

$$C_{\text{reducible}} \geqslant \sum_{t \in \mathcal{R}} \sum_{j=0}^{4b-1} C^t_{MC_j} - \mathcal{D} \quad \text{and} \quad C_{\text{reducible}} \geqslant 0, \tag{7}$$

and the number of conditions which can not be controlled by the attacker

$$C_{\text{trunc}} = \sum_{t \in \mathbb{Z}_{T \cdot m} \setminus \mathcal{R}} \sum_{j=0}^{4b-1} C^t_{MC_j}. \tag{8}$$

The goal is now to find the minimal attack costs by solving this MILP model

$$\textbf{minimize:} \quad C_{\text{collision}} + C_{\text{trunc}} + C_{\text{reducible}}. \tag{9}$$

If we do not allow the attacker to utilize any degrees of freedom, the parameters $T = 4$ and $m = 2$ would be sufficient for Haraka-512 v2, and parameters $T = 2$ and $m = 2$ would suffice for Haraka-256 v2 (see Table 2). However, as discussed in Section 3.2, this approach would be too optimistic (from our perspective). Taking into account the assumptions we make on the capabilities of an attacker utilizing the degrees of freedom, at least $T = 5$ rounds are required (see Table 4) for the best attack to require at least 2^{256} steps.

[6] Using Gurobi 6.5.0 (linux64), Intel(R) Core(TM) i7-4770S CPU @ 3.10GHz, 16GB RAM

Table 2: Complexity bounds ($\log_2$) of the best attack in our truncated setting, over multiple rounds, without utilizing degrees of freedom

(a) Security for π_{512}

	m				
T	1	2	3	4	5
1	0	32	48	64	64
2	32	128	96	96	96
3	48	192	176	192	192
4	112	256	256	256	256
5	128	256	256	256	256
6	208	256	256	256	256
7	224	256	256	256	256

(b) Security for π_{256}

	m				
T	1	2	3	4	5
1	0	0	0	0	128
2	0	256	176	192	192
3	184	256	240	256	256
4	176	256	256	256	256
5	256	256	256	256	256
6	240	256	256	256	256
7	256	256	256	256	256

Table 4: Complexity bounds ($\log_2$) of the the best attack in our truncated setting, utilizing additional degrees of freedom over $2q$ rounds for π_{512} and π_{256}, with $m = 2$ fixed. Entries which are bold are not better then the generic attacks.

(a) Second-preimage for π_{512}

T	1	2	3	4	5	6
$q = 1$	0	96	144	**256**	**256**	**256**
$q = 2$	0	0	96	128	**256**	**256**
$q = 3$	0	0	0	96	128	**256**

(b) Collision for π_{512}

T	1	2	3	4	5	6
$q = 1$	0	48	**136**	**176**	**256**	**256**
$q = 2$	0	0	40	96	**168**	**256**
$q = 3$	0	0	0	32	96	**160**

(c) Second-preimage for π_{256}

T	1	2	3	4	5	6
$q = 1$	0	176	192	**256**	**256**	**256**
$q = 2$	0	128	128	192	**256**	**256**
$q = 3$	0	0	128	128	192	**256**

(d) Collision for π_{256}

T	1	2	3	4	5	6
$q = 1$	0	**168**	**176**	**240**	**256**	**256**
$q = 2$	0	64	112	**160**	**256**	**256**
$q = 3$	0	0	64	112	**176**	**256**

In Figure 1, we give an example to illustrate how this attack model works for a collision attack. The attacker starts in this case at S^4 and can control $q = 2$ rounds in both directions. When searching for a collision, the attacker has control over the full state, therefore he has enough degrees of freedom available to fulfill the conditions for the transitions through MixColumns in the controlled rounds. The only remaining part is the transition in the first round which happens with a probability of 2^{-16}.

Meet-in-the-middle Attacks

A powerful technique for finding preimages are meet-in-the-middle techniques and they have been applied to various AES-based hash functions, for instance Whirlpool [44] and Grøstl [50]. The basic attack principle is to

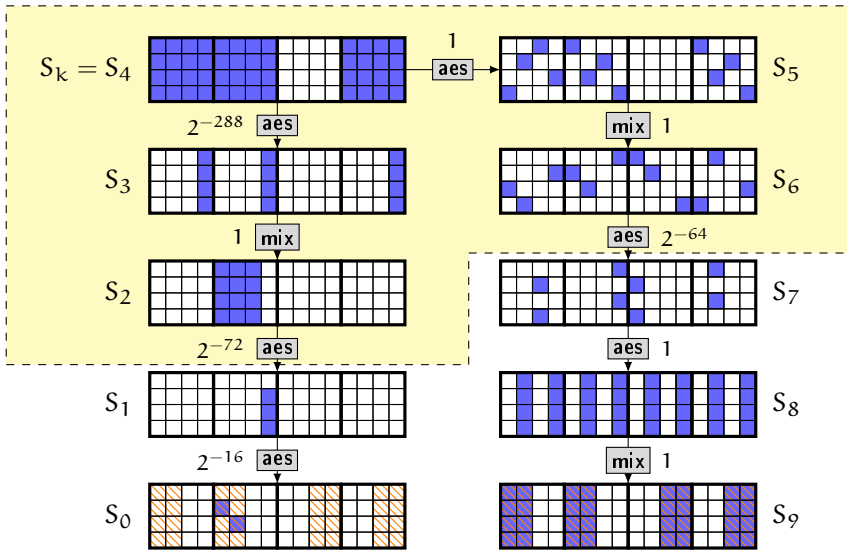


Figure 1: Truncated model utilizing degrees of freedom for $T = 3, m = 2$ and $q = 2$. An active byte is marked as ■; a byte which is removed due to `trunc` is marked with ▨; boxes [f] denotes a function f mapping one state to the next, and the number next to it gives the transition probability. For finding a collision, the attacker would have full control over the middle rounds, marked in the highlighted area. As there are only 53 conditions on bytes which all can be fulfilled with the available degrees of freedom the attack costs for an idealized attacker would be 2^{16}.

split the function into two sub-functions, such that a part of the message only affects the first function and another part of the message the second function. These sub-functions are referred to as chunks (of rounds) and bytes which only affect one them are called *neutral* bytes. The limiting constraint of this attack is the number of rounds we can independently propagate our message through these chunks.

We are interested in finding out the highest number of rounds of Haraka-256 v2 and Haraka-512 v2 that can be attacked. In this case, the strategy is to have a single neutral byte in the forward and backward chunk. We can check for all possible positions of two unknown bytes after how many rounds we still are able to find a match, meaning that the state is not unknown in all bytes. Starting at the beginning of a Haraka v2 round in the forward direction we can still compute the value of 16 bytes after $\mathsf{SR} \circ \mathbf{mix} \circ \mathsf{MC} \circ \mathsf{SR} \circ \mathsf{MC} \circ \mathsf{SR}$. In the backwards direction we can also compute 16 bytes after $\mathsf{SR}^{-1} \circ \mathsf{MC}^{-1} \circ \mathbf{mix}^{-1} \circ \mathsf{SR}^{-1} \circ \mathsf{MC}^{-1} \circ \mathsf{SR}^{-1} \circ \mathsf{MC}^{-1} \circ \mathbf{mix}^{-1}$. In total this covers 3 rounds of

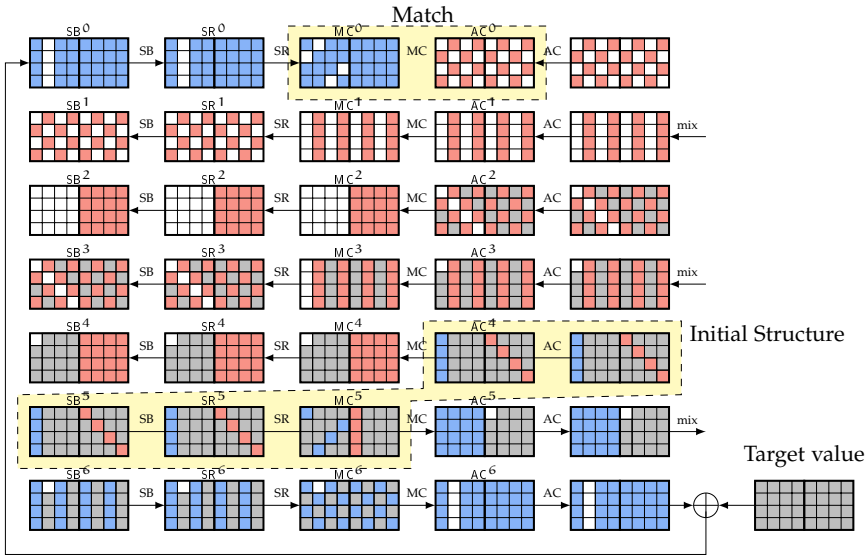


Figure 2: Meet-in-the-middle attack on 3.5 rounds of Haraka-256 v2. All □ are unknown, ▒ are constant, ▒ neutral bytes backward and ▒ neutral bytes forward.

Haraka v2. In an attack we can choose a different starting point, but the total number of rounds which can be covered stays the same. The initial structure technique [45] allows us to further extend the separation of the two chunks by 1 round.

We can use this now to mount an attack on 3.5 rounds of Haraka-256 v2, following the procedure given in [44] (see Figure 2):

1. Randomly select values for the constant bytes ▒ in AC^4.
2. For all 2^8 possible choices for $\mathsf{AC}^4_{*,0}$ ▒ which keep $\mathsf{MC}^4_{0,0}, \mathsf{MC}^4_{1,0}, \mathsf{MC}^4_{2,0}$ constant, compute forward to obtain the state in MC^0 and store the result in a table T.
3. For all 2^8 possible choices for $\mathsf{MC}^5_{*,4}$ ▒ which keep $\mathsf{AC}^5_{0,4}, \mathsf{AC}^5_{1,4}, \mathsf{AC}^5_{2,4}$ constant, compute backward to obtain the state in AC^0.
4. Check if there is an entry in T that matches with AC^0 through MixColumns. If so check whether the remaining bytes also match, otherwise repeat from step 2 (or step 1 if necessary).

Matching. We can check whether the states MC^0 and AC^0 can be matched through MixColumns in the following way. Lets consider the first column, which gives us the following two equations

$$\mathsf{AC}^0_{2,0} = \mathsf{MC}^0_{3,0} \oplus 2 \cdot \mathsf{MC}^0_{2,0} \oplus 3 \cdot \mathsf{MC}^0_{1,0} \oplus \mathsf{MC}^0_{0,0} \tag{10}$$

$$\mathsf{AC}^0_{0,0} = 3 \cdot \mathsf{MC}^0_{3,0} \oplus \mathsf{MC}^0_{2,0} \oplus \mathsf{MC}^0_{1,0} \oplus 2 \cdot \mathsf{MC}^0_{0,0} \tag{11}$$

As we know the values for $\mathsf{MC}^0_{3,0}, \mathsf{MC}^0_{1,0}, \mathsf{MC}^0_{0,0}$ we can simplify this to

$$\mathsf{AC}^0_{2,0} \oplus C_0 = 2 \cdot \mathsf{MC}^0_{2,0} \quad \text{and} \quad \mathsf{AC}^0_{0,0} \oplus C_1 = \mathsf{MC}^0_{2,0}. \tag{12}$$

We can use this now to check whether we can fulfill:

$$\mathsf{AC}^0_{2,0} \oplus C_0 = 2 \cdot (\mathsf{AC}^0_{0,0} \oplus C_1) \tag{13}$$

$$\mathsf{AC}^0_{2,0} \oplus 2 \cdot \mathsf{AC}^0_{0,0} = C_0 \oplus 2 \cdot C_1, \tag{14}$$

where the right side can be computed in step (2) and the left side in step (3) of our attack.

Complexity. Computing the table T and 2^8 values for AC^0 costs 2^8 3.5-round Haraka-256 v2 evaluations and requires $2^8 \cdot 8$ bytes of memory, as we only need to store 1 byte of information for each column. The success probability for the match is 2^{-32} for the left half of the state and 2^{-64} for the right half. Hence, on average $2^8 \cdot 2^8 \cdot 2^{-96} = 2^{-80}$ candidates will remain in Step 4. There are still $12 + 8$ byte conditions which have to be satisfied, therefore if we repeat step 1-4 2^{240} times we expect to find $2^{240} \cdot 2^{-80} \cdot 2^{-20 \cdot 8} = 1$ solution. The overall complexity is $2^{240} \cdot 2^8 = 2^{248}$ evaluations of Haraka v2 to find a preimage.

We were not able to extend the attack to 4 rounds, as we would have only two bytes in each column of MC^0 and AC^0. In this case we can not filter out solutions in the matching step. For Haraka-512 v2 we can attack 4 rounds in a very similar way (see Section D).

Attack on Haraka v1 by Jean

An attack by Jean [30] on a previous version of Haraka v2, denoted Haraka v1, has been published. In this section we explain how the attack was possible, and why it is not applicable to Haraka v2 presented in this paper. In [37] it was shown that if the two halves of an AES state are equal, then applying a keyless AES round function preserves this property. In Haraka v1, round constants exhibited strong symmetries in the sense that i) the same constant was used for each block, and ii) the same constant was used for each column

in each block. The observation by Jean is, that when using the property of [37] together with the weak constants and the fact that the mixing layer of Haraka v2 permute the columns, one can construct an efficient structural distinguisher that allows for collisions and preimages.

In Haraka v2 presented in this paper, this structural property has been dealt with by destroying properties (i) and (ii) above, particularly by using round constants based on the digits of π. We refer to Section 2 for the details. We remark that the new choice of constants *do not affect* the performance of Haraka v2. As the attack was structural, and feasible purely due to the round constants, we believe the number of rounds for Haraka v2, which is based on the truncated model (see Section 4.2), is still well-founded and provides long-term security.

4.3 Collision Resistance

While we explicitly do not require collision resistance for Haraka v2, we still discuss the security level with respect to this criteria in the following. Similar to our arguments for second-preimage security, we can apply our truncated model for finding collisions. The best collision attacks on AES-based hash functions are based on the rebound attack, and these are covered by our model. However, for finding a collision, an attacker can freely choose the complete internal state and not only the differences. This translates to more degrees of freedom. Therefore, the expected security level is lower for the same number of rounds (see Table 4).

The best generic attack has a lower complexity of 2^{128} compared to the second-preimage case, which might suggest that one only requires 2^{128} in our truncated security model. However, this would still indicate some non-ideal property, and it is likely that the more relaxed collision setting allows to exploit this after using up all degrees of freedom. Consequently, we opt to also aim for a security level of 2^{256} in our truncated security model, which requires adding one round for Haraka-512 v2.

4.4 Design Choices

In the following, we interpret the security analysis of Section 4.2 which led to the proposed parameters and design choices. We recall that T denotes the number of rounds of either π_{512} or π_{256}, and m denotes the number of AES rounds applied to each of the b blocks in each round.

Mode

As described, we use the DM mode for our permutations to define Haraka v2. Other modes were considered, including a sponge-based construction and a block cipher in DM mode. The choice to use a permutation in DM mode is motivated both by performance and security considerations. We refer to Section E for the details.

Round Parameters T and m

One of the first questions which arise is how the number of AES rounds and frequency of mixing the individual states influences the security bounds. Our analysis of Table 1 gives a strong indication that $m = 2$ is an optimal choice, as it gives the best trade-off between number of active S-boxes and the total number of required AES rounds Tmb. The number of rounds is chosen as $T = 5$, as this gives the required security parameters in the truncated model, even when assuming a powerful attacker controlling more rounds than the best known attacks are capable of.

Mixing Layers

For the mixing layer, a variety of choices were considered. Our main criteria were that the layer should be efficiently implementable (see Section 5.2) on our target platforms, while still contributing to a highly secure permutation. Other potential candidates for the mixing layer are discussed in Section E. With respect to our criteria, for most choices of T and m, using the proposed $\mathbf{mix}_{512}$ and $\mathbf{mix}_{256}$ give a significantly higher number of active S-boxes, compared to other approaches discussed in Section E.

Truncation Pattern for Haraka-512 v2

There are many possible choices for the truncation pattern for Haraka-512 v2. In our analysis, we consider truncation patterns which truncate row-wise or column-wise, as these are most efficient to implement, due to the way words are stored in memory. The pattern we chose is taking the two least significant columns of the first two blocks and the two most significant columns of the last two states. We found that this approach compared favorably, with respect to the number of active S-boxes, to row-wise patterns or patterns choosing the same two columns from each state.

Table 6: Latency and inverse throughput for one-round AES instructions on target platforms

Architecture	L_{aes} [cycles]	T_{aes}^{-1} [instructions/cycle]
Haswell	7	1
Skylake	4	1

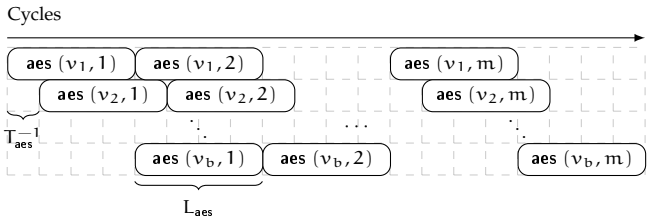


Figure 1: Pipelined AES instructions. A box **aes** (v, i) denotes the application of the ith AES round to a block v.

5 Implementation Aspects and Performance

As mentioned, Haraka v2 is designed *solely for use on platforms with AES hardware support*. To that end, we assume the existence of a hardware instruction pipeline, which can execute a single round of the AES with an instruction denoted **aes**, with a *latency* of L_{aes} cycles and an *inverse throughput* of T_{aes}^{-1} instructions per cycle (given for our target architectures in Table 6). We remark that our Haswell test machine has an i7-4600M CPU at 2.90GHz; the Skylake machine has an i7-6700 CPU at 3.40GHz. We furthermore expect Haraka v2 to be efficiently implementable on ARMv8 due to its support of AES instructions. We remark that the Turbo Boost technology has been switched off for all our performance measurements.

When encrypting a single block with the AES, one must wait L_{aes} cycles each time the block is encrypted for one round. However, if the inverse throughput T_{aes}^{-1} is low compared to L_{aes}, and if additional *independent* data blocks are available for processing, one can use this data independency to better utilize the AES pipeline. Thus, in theory, if using $k = L_{aes} \cdot T_{aes}^{-1}$ independent blocks for the AES, one can encrypt each of those blocks for a single round in just $(k-1) \cdot T_{aes}^{-1} + L_{aes}$ cycles, while m rounds of the AES can be completed for all k blocks in just $(k-1) \cdot T_{aes}^{-1} + L_{aes} \cdot m$ cycles, as illustrated in Figure 1. As such, with Haraka v2 using several AES blocks, the pipeline is better utilized.

5.1 Multiple Inputs

As described above, the theoretically optimal choice of state blocks, performance wise, would be $b = L_{\mathsf{aes}} \cdot T_{\mathsf{aes}}^{-1}$. However, Haraka v2 uses a varying number of blocks. To that end, we consider for both Haraka-512 v2 and Haraka-256 v2 the parallel application of the corresponding function to *multiple inputs*, assuming that such are available for processing. For example, if $k = L_{\mathsf{aes}} \cdot T_{\mathsf{aes}}^{-1} = 7$, with a state size of $b = 4$ blocks, one could process two *independent* inputs x and x' in parallel, thus artificially extending the state to $b = 8$ blocks, allowing better pipeline utilization. We denote the number of parallel inputs processed in this manner by P. For each of our constructions and target platforms, there will be an optimal choice of P which allows good AES pipeline utilization while, at the same time, keeping the full context in low-level cache.

5.2 Implementation of Linear Mixing

Consider the case where $P = 1$, i.e. when using a single input. Even if the number of blocks in the state is less than $L_{\mathsf{aes}} \cdot T_{\mathsf{aes}}^{-1}$, a number of the instructions used for the linear mixing can be hidden after the **aes** operation. For example, while the instruction to encrypt the second AES round of a Haraka v2 round is still being executed for one or more blocks, while other blocks have already finished, instructions pertaining to the mixing of the finished blocks can be executed while the AES instructions for the remaining blocks are allowed to finish. To that end, more so than otherwise, choosing instructions for the linear mixing layer with low latency and high throughput is important.

For the implementation of $\mathbf{mix}_{512}$ and $\mathbf{mix}_{256}$, we make use of the `punpckhdq` and `punpckldq` instructions. On both Haswell and Skylake, those instructions have a latency of 1 clock cycle and an inverse throughput of 1 instruction/cycle. In the case of Haraka-512 v2, where the state has $b = 4$ blocks, $\mathbf{mix}_{512}$ uses eight instructions in the mixing layer, while for Haraka-256 v2 we require just one call to each of the instructions.

5.3 Haraka v2 Performance and Discussion

In the following, we present the performance of Haraka v2 when implemented on the Haswell and Skylake platforms, and discuss their performance in relation to other primitives which would be other potential candidates for our target applications.

First of all, it is interesting to compare Haraka v2 to SPHINCS-256-H and SPHINCS-256-F from the SPHINCS-256 construction [9], which have identi-

Table 7: Benchmarks for various primitives on the Haswell and Skylake platforms. We give the implementation type as well as state size, block size and output size. Implementations marked † are taken from SUPERCOP (see eBACS [8]); the rest are written by us. For selected primitives, we give the performance using a varying number of independent inputs processed in parallel, $P \in \{1, 4, 8\}$.

			Sizes (bits)				
	Primitive	Implementation	State	Block	Output	Haswell	Skylake
$P = 1$	**Haraka-256 v2**	AES-NI	256	256	256	**1.25**	**0.72**
	Haraka-512 v2	AES-NI	512	512	256	**1.75**	**0.97**
	AESQ (from PAEQ)	AES-NI†	512	512	512	3.75	2.19
	Simpirav2[b = 2]	AES-NI	256	256	256	2.59	1.94
	Simpirav2[b = 4]	AES-NI	512	512	512	4.47	2.38
	SPHINCS-256-H	AVX2†	512	256	256	11.16	10.92
	SPHINCS-256-F	AVX2†	512	256	256	11.31	11.12
$P = 4$	**Haraka-256 v2**	AES-NI	1024	256	1024	1.12	0.63
	Haraka-512 v2	AES-NI	2048	512	1024	1.38	0.72
	Simpirav2[b = 2]	AES-NI	2048	512	1024	2.37	1.62
	Simpirav2[b = 4]	AES-NI	2048	512	1024	2.03	1.17
$P = 8$	**Haraka-256 v2**	AES-NI	2048	256	2048	1.14	0.66
	Haraka-512 v2	AES-NI	4096	512	2048	1.43	0.92
	Simpirav2[b = 2]	AES-NI	2048	256	2048	1.87	1.18
	Simpirav2[b = 4]	AES-NI	4096	512	4096	1.56	1.35
	SPHINCS-256-H	AVX2†	4096	256	2048	1.99	1.62
	SPHINCS-256-F	AVX2†	4096	256	2048	2.11	1.71

cal functional signatures and similar design criteria to Haraka-512 v2 and Haraka-256 v2 respectively. If we first consider the performance using 8-way parallelization (i.e. using $P = 8$), we see from Table 7 that the SPHINCS functions have a performance of 1.62 cpb on Skylake for the H function and 1.71 cpb for the F function. These implementations do not utilize AVX-512 (employing 512-bit registers), so it is reasonable to assume their performance could be doubled under such circumstance. However, we note that even under this assumption, in both the cases of Haswell and Skylake, Haraka v2 performs favorably in comparison to those of SPHINCS-256.

In some applications, including some of the function calls in hash-based signatures, several calls to the short-input hash function *can not be parallelized*. To that end, it is of interest to compare the performance for Haraka v2, using $P = 1$, to the corresponding functions from SPHINCS-256. In this case, from the first part of Table 7, we see that Haraka-256 v2 performs very well with 1.25 cpb and 0.72 cpb on Haswell and Skylake, respectively, while the numbers for Haraka-512 v2 are 1.75 cpb on Haswell and 0.97 cpb on Skylake. From benchmarking the corresponding SPHINCS-256 functions on the same machines using $P = 1$, we obtain a performance of 11.16 cpb on Haswell and 10.92 cpb on Skylake for their H function, and respectively 11.31 and 11.12 cpb for their F function on Haswell and Skylake respectively. Thus,

when the hash function calls in hash-based signatures can not be parallelized, Haraka v2 performs between 6.5 times and 15 times better on our tested platforms.

In Table 7, we compare the performance of Haraka v2 not only with the SPHINCS-256 functions, but also other designs which exhibit similar block-, input- and output sizes as Haraka v2. We comment on their benchmarks in the following.

With AESQ and Haraka v2 having very similar designs, the former having 20 AES rounds per block compared to Haraka v2 with 10, it is reasonable that AESQ is about twice as slow as Haraka-512 v2. The remaining margin can be accredited to AESQ employing an evolving round key update rather than tabularized constants like Haraka v2.

Another close competitor is Simpira v2, which we implemented and benchmarked using $b = 2$ and $b = 4$, i.e. with two and four AES blocks respectively, thereby matching the sizes of the Haraka-256 v2 and Haraka-512 v2. Simpira v2 uses 15 AES rounds per block. Despite this being less than AESQ, it is slower when $P = 1$, because only one AES round (for $b = 2$) or two AES rounds (for $b = 4$) can be computed in parallel due to the Feistel structure. This is confirmed when we consider the performance of Simpira v2 with $P = 4$ and $P = 8$. For $b = 2$, parallelizing over $P = 4$ inputs brings the performance up to 2.37 cpb on Haswell and 1.62 cpb on Skylake. For $b = 4$, the performance is boosted up to 2.03 cpb and 1.17 cpb for $P = 4$ on Haswell and Skylake respectively. With $P = 8$, the effect of parallelization brings its performance up to 1.56 cpb for Haswell and 1.35 cpb for Skylake. We remark that in the Simpira v2 paper [22], the authors give their own benchmarks using $P = 4$ independent inputs. They report performance slightly better than ours, at 0.95 cpb for $b = 2$ and 0.94 cpb for $b = 4$ measured on a Skylake machine. However, as no source code was provided, we wrote our own optimized implementation.

From Table 7, we see that when multiple independent inputs are available for processing, the gap between the performance of Haraka v2 and of the Simpira v2 and SPHINCS functions diminishes. This makes sense as essentially processing multiple inputs gives a source of independence to draw on, allowing to parallelize instruction calls which would not otherwise be possible. As such, the throughput becomes more a question of the total number of instructions needed to obtain the desired security level, and less about the interplay of these instructions. With Haraka v2 still performing favorably, there are a couple of interesting observations. First, Haraka v2 performs better with $P = 4$ than with $P = 8$. This is simply due to the number of 128-bit registers available; with $P = 8$ more overhead occurs due to otherwise unnecessary read/write operations. Simpira v2 with its fewer parallel AES round applica-

tions in general need to process more independent inputs to achieve its optimal performance, as is evident from Table 7. Second, the best performance obtained overall is 0.63 cpb on Skylake with Haraka-256 v2 for $P = 4$. This matches very well with the theoretical maximum of $(20 \cdot 4)/(32 \cdot 4) = 0.625$ cycles per processed byte.

We considered also comparing against the recent KangarooTwelve extendable output hash from the Keccak team [10]. It uses 12 rounds of the Keccak permutation in a sponge construction, employing tree hashing when possible. However, for a short input of only 64 bytes, only one permutation call is needed and no parallelization can be made. The authors state a latency of ≈ 530 cycles in this case, yielding 8.28 cpb for a 512-bit input and half that performance for a 256-bit input, even when using Skylake-optimized implementations. In the Simpira v2 paper, the authors compare against an optimized SHA-256 implementation which is parallelized for $P = 4$ "long" inputs, claiming a performance of 2.35 cpb. For short inputs, and also considering the $P = 1$ cases, the performance would be reduced to a fraction of that, excluding also SHA-256 as a competitor. However, as no source code was provided, we were not able to verify against our own benchmarks. With this said, generic hash functions generally obtain their best performance for longer input, and for most such functions their poor performance on short inputs come as no surprise.

5.4 Performance of SPHINCS using Haraka v2

While the previous performance figures provide a good comparison between the functions themselves, the actual performance figures relevant for a hash-based signature scheme are the costs for key generation, signing and verifying a signature. The total costs for these operations are difficult to derive by only looking at the performance of the short-input hash function. For that reason, we modified the optimized AVX implementation of SPHINCS given in [9], by replacing all calls to SPHINCS-256-F and SPHINCS-256-H by our implementations of Haraka-256 v2 and Haraka-512 v2 respectively. Parallel calls to these functions are processed to the same extent, using $P = 8$ calls at the same time, and no further optimizations have been applied. As can be seen in Table 8, the current performance gains by using Haraka v2 are between a factor of 1.50 to 2.86, depending on the platform and operation.

6 Conclusion and Remarks on Future Work

Together with in-depth implementation considerations on CPUs offering AES hardware acceleration, we presented the seemingly fastest proposal

Table 8: Comparison of the AVX implementation of SPHINCS-256 with our implementation using Haraka v2. All numbers are given as the total number of cycles required, and are measured using SUPERCOP. The speed-up factor of operations are given in parentheses.

	Haswell		Skylake	
	SPHINCS-256	Haraka v2	SPHINCS-256	Haraka v2
Key generation	3,295,808	2,060,864 (×1.60)	2,839,018	1,426,138 (×1.99)
Signing	52,249,518	34,938,076 (×1.50)	43,517,538	23,312,354 (×1.87)
Verification	1,495,416	695,222 (×2.15)	1,291,980	452,066 (×2.86)

for compression resp. short-input hashing on our target platforms, with a performance of less than 1 cpb on a Skylake desktop CPU, both with and without parallelization across multiple inputs. As a concrete application of Haraka v2, we show that by using it inside the SPHINCS-256 hash-based signature, we can speed up the key generation, signing and verification operations by factors $\times 1.99$, $\times 1.87$ and $\times 2.86$, respectively, on Intel's Skylake architecture.

Despite having explored a larger design-space, Haraka v2 ended up having strong similarities with the AESQ permutation, used in the CAESAR candidate PAEQ [11]. All implementations for Haraka v2, including code for security analysis and for SPHINCS using Haraka v2, are publicly available[7].

We cover the important differential- and meet-in-the-middle attack vectors in our security analysis. We also give security arguments for Haraka v2 against various classes of attacks, without having to treat a large part of the hash function as a black box, as is the usual approach. This, of course, does not rule out attacks outside of the models that we consider. Hence, as for all other cryptographic primitives, more cryptanalysis is useful to establish more trust in the proposal.

Returning to the question: How much faster can a hash function become, if collision resistance is dropped from the list of requirements? With Haraka v2, we drop from $T = 6$ rounds to $T = 5$ rounds and still retain security against second-preimage attacks. We conclude that the performance gains are limited for the class of strategies considered, namely AES-like designs. This particularly holds when aiming at pre-quantum security levels higher than those for collision resistance, namely 256 bits rather than 128 bits. Aiming at higher security levels make sense, as there is evidence (at least for generic attacks), that the post-quantum security level will be 128 bits in both cases. Of course, this argument does not consider non-generic attacks that use capabilities of

[7] See supplementary material

hypothetical quantum computers, and we leave investigations in this direction as future work.

References

[1] Jean-Philippe Aumasson and Daniel J. Bernstein. "SipHash: A Fast Short-Input PRF." In: *Progress in Cryptology - INDOCRYPT*. 2012, pp. 489–508.

[2] Nasour Bagheri, Florian Mendel, and Yu Sasaki. "Improved Rebound Attacks on AESQ: Core Permutation of CAESAR Candidate PAEQ." In: *Information Security and Privacy, ACISP 2016*. 2016.

[3] Mihir Bellare. "New Proofs for NMAC and HMAC: Security without Collision Resistance." In: *J. Cryptology* 28.4 (2015), pp. 844–878.

[4] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying Hash Functions for Message Authentication." In: *CRYPTO '96*. 1996, pp. 1–15.

[5] Mihir Bellare and Phillip Rogaway. "Collision-Resistant Hashing: Towards Making UOWHFs Practical." In: *CRYPTO '97*. 1997, pp. 470–484.

[6] Ryad Benadjila, Olivier Billet, Henri Gilbert, Gilles Macario-Rat, Thomas Peyrin, Matt Robshaw, and Yannick Seurin. *SHA-3 Proposal: ECHO*. Submission to NIST (updated). 2009.

[7] Daniel J. Bernstein. *Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete?* https://cr.yp.to/hash/collisioncost-20090823.pdf. 2009.

[8] Daniel J. Bernstein and Tanja Lange (editors). *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. http://bench.cr.yp.to, accessed 19 November 2015.

[9] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. "SPHINCS: Practical Stateless Hash-Based Signatures." In: *EUROCRYPT 2015*. 2015, pp. 368–397.

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *KangarooTwelve: fast hashing based on Keccak-p*. URL: \url{http://keccak.noekeon.org/KangarooTwelve.pdf}.

[11] Alex Biryukov and Dmitry Khovratovich. *PAEQ*. Submission to the CAESAR competition. https://competitions.cr.yp.to/round1/paeqv1.pdf. 2014.

[12] Gilles Brassard, Peter Høyer, and Alain Tapp. "Quantum Cryptanalysis of Hash and Claw-Free Functions." In: *LATIN '98: Theoretical Informatics*. 1998, pp. 163–169. DOI: 10.1007/BFb0054319.

[13] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. "XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions." In: *Post-Quantum Cryptography - PQCrypto 2011*. 2011, pp. 117–129.

[14] Jiali Choy, Huihui Yap, Khoongming Khoo, Jian Guo, Thomas Peyrin, Axel Poschmann, and Chik How Tan. "SPN-Hash: Improving the Provable Resistance against Differential Collision Attacks." In: *AFRICACRYPT 2012*. 2012, pp. 270–286.

[15] Scott Contini, Arjen K. Lenstra, and Ron Steinfeld. "VSH, an Efficient and Provable Collision-Resistant Hash Function." In: *EUROCRYPT 2006*. 2006, pp. 165–182.

[16] Christophe De Canniere, Hisayoshi Sato, and Dai Watanabe. *Hash Function Luffa: Supporting Document*. Submission to NIST (Round 1). 2008.

[17] Itai Dinur and Adi Shamir. "An Improved Algebraic Attack on Hamsi-256." In: *Fast Software Encryption, FSE 2011*. 2011, pp. 88–106.

[18] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. *Cryptanalysis of Simpira v1*. Cryptology ePrint Archive, Report 2016/244. http://eprint.iacr.org/2016/244. 2016.

[19] Thomas Fuhr. "Finding Second Preimages of Short Messages for Hamsi-256." In: *ASIACRYPT 2010*. 2010, pp. 20–37.

[20] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. *Grøstl – a SHA-3 candidate*. Submission to NIST (Round 3). 2011. URL: http://www.groestl.info/Groestl.pdf.

[21] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search." In: *ACM Symposium on the Theory of Computing*. 1996, pp. 212–219. DOI: 10.1145/237814.237866.

[22] Shay Gueron and Nicky Mouha. *Simpira v2: A Family of Efficient Permutations Using the AES Round Function*. Cryptology ePrint Archive, Report 2016/122. 2016.

[23] Jian Guo, Thomas Peyrin, and Axel Poschmann. "The PHOTON Family of Lightweight Hash Functions." In: *CRYPTO 2011*. 2011, pp. 222–239.

[24] Shai Halevi, William E. Hall, and Charanjit S. Jutla. *The Hash Function Fugue*. Submission to NIST (updated). 2009. URL: http://domino.research.ibm.com/comm/research_projects.nsf/pages/fugue.index.html/$FILE/fugue_09.pdf.

[25] Andreas Hülsing, Joost Rijneveld, and Fang Song. "Mitigating Multi-target Attacks in Hash-Based Signatures." In: *Public-Key Cryptography - PKC 2016*. 2016, pp. 387–416. DOI: 10.1007/978-3-662-49384-7_15.

[26] IBM. *ILOG CPLEX Optimizer*. 2016. URL: \url{http://www-01.ibm.com/software/commerce/optimization/}.

[27] Gurobi Optimization Inc. *Gurobi Optimizer Reference Manual*. 2016. URL: \url{http://www.gurobi.com}.

[28] Sebastiaan Indesteege. *The LANE hash function*. Submission to NIST. 2008. URL: http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf.

[29] Jérémy Jean, María Naya-Plasencia, and Thomas Peyrin. "Improved Cryptanalysis of AES-like Permutations." In: *J. Cryptology* 27.4 (2014), pp. 772–798.

[30] Jérémy Jean. *Cryptanalysis of Haraka*. Cryptology ePrint Archive, Report 2016/396. 2016.

[31] Jérémy Jean and Ivica Nikolic. "Efficient Design Strategies Based on the AES Round Function." In: *Fast Software Encryption, FSE 2016*. 2016.

[32] Elif Bilge Kavun, Martin M. Lauridsen, Gregor Leander, Christian Rechberger, Peter Schwabe, and Tolga Yalçın. *Prøst*. Submission to the CAESAR competition. https://competitions.cr.yp.to/round1/proestv1.pdf. 2014.

[33] John Kelsey and Bruce Schneier. "Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work." In: *EUROCRYPT 2005*. 2005, pp. 474–490.

[34] Lars R. Knudsen. "Truncated and Higher Order Differentials." In: *Fast Software Encryption, FSE 1994*. 1994, pp. 196–211.

[35] Özgül Küçük. *The Hash Function Hamsi*. Submission to NIST (updated). 2009. URL: http://www.cosic.esat.kuleuven.be/publications/article-1203.pdf.

[36] Leslie Lamport. *Constructing digital signatures from a one way function*. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory. 1979.

[37] Tri Van Le, Rüdiger Sparr, Ralph Wernsdorf, and Yvo Desmedt. "Complementation Like and Cyclic Properties of AES Round Functions." In: *Advanced Encryption Standard - AES, 4th International Conference, AES 2004*. 2004, pp. 128–141. DOI: 10.1007/11506447_11.

[38] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. "The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl." In: *Fast Software Encryption, FSE 2009*. 2009, pp. 260–276.

[39] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. "Differential and Linear Cryptanalysis Using Mixed-Integer Linear Programming." In: *Inscrypt 2011*. 2011, pp. 57–76.

[40] Ivica Nikolić. *Tiaoxin*. Submission to the CAESAR competition. https://competitions.cr.yp.to/round1/paeqv1.pdf. 2014.

[41] PQCRYPTO EU Project. *Paris workshop, November 2015*.

[42] Christian Rechberger. "Second-Preimage Analysis of Reduced SHA-1." In: *ACISP 2010*. 2010, pp. 104–116.

[43] Sondre Rønjom. "Invariant subspaces in Simpira." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 248. URL: http://eprint.iacr.org/2016/248.

[44] Yu Sasaki. "Meet-in-the-Middle Preimage Attacks on AES Hashing Modes and an Application to Whirlpool." In: *Fast Software Encryption - 18th International Workshop, FSE 2011*. 2011, pp. 378–396.

[45] Yu Sasaki and Kazumaro Aoki. "Finding Preimages in Full MD5 Faster Than Exhaustive Search." In: *Advances in Cryptology - EUROCRYPT 2009*. 2009, pp. 134–152.

[46] Yu Sasaki, Yang Li, Lei Wang, Kazuo Sakiyama, and Kazuo Ohta. "Non-full-active Super-Sbox Analysis: Applications to ECHO and Grøstl." In: *ASIACRYPT 2010*. 2010, pp. 38–55.

[47] Siwei Sun, Lei Hu, Peng Wang, Kexin Qiao, Xiaoshuang Ma, and Ling Song. "Automatic Security Evaluation and (Related-key) Differential Characteristic Search: Application to SIMON, PRESENT, LBlock, DES(L) and Other Bit-Oriented Block Ciphers." In: *ASIACRYPT 2014*. 2014, pp. 158–178.

[48] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. "Finding Collisions in the Full SHA-1." In: *CRYPTO 2005*. 2005, pp. 17–36.

[49] Hongjun Wu and Bart Preneel. *AEGIS*. Submission to the CAESAR competition. https://competitions.cr.yp.to/round1/aegisv1.pdf. 2014.

[50] Shuang Wu, Dengguo Feng, Wenling Wu, Jian Guo, Le Dong, and Jian Zou. "(Pseudo) Preimage Attack on Round-Reduced Grøstl Hash Function and Others." In: *Fast Software Encryption, FSE 2012*. 2012, pp. 127–145.

[51] Hongbo Yu, Gaoli Wang, Guoyan Zhang, and Xiaoyun Wang. "The Second-Preimage Attack on MD4." In: *CANS 2005*. 2005, pp. 1–12.

A Round Constants

Table 9: Round constants used in π_{512} and π_{256}.

RC_0	0684704ce620c00ab2c5fef075817b9d	RC_{20}	d3bf9238225886eb6cbab958e51071b4
RC_1	8b66b4e188f3a06b640f6ba42f08f717	RC_{21}	db863ce5aef0c677933dfddd24e1128d
RC_2	3402de2d53f28498cf029d609f029114	RC_{22}	bb606268ffeba09c83e48de3cb2212b1
RC_3	0ed6eae62e7b4f08bbf3bcaffd5b4f79	RC_{23}	734bd3dce2e4d19c2db91a4ec72bf77d
RC_4	cbcfb0cb4872448b79eecd1cbe397044	RC_{24}	43bb47c361301b434b1415c42cb3924e
RC_5	7eeacdee6e9032b78d5335ed2b8a057b	RC_{25}	dba775a8e707eff603b231dd16eb6899
RC_6	67c28f435e2e7cd0e2412761da4fef1b	RC_{26}	6df3614b3c7559778e5e23027eca472c
RC_7	2924d9b0afcacc07675ffde21fc70b3b	RC_{27}	cda75a17d6de7d776d1be5b9b88617f9
RC_8	ab4d63f1e6867fe9ecdb8fcab9d465ee	RC_{28}	ec6b43f06ba8e9aa9d6c069da946ee5d
RC_9	1c30bf84d4b7cd645b2a404fad037e33	RC_{29}	cb1e6950f957332ba25311593bf327c1
RC_{10}	b2cc0bb9941723bf69028b2e8df69800	RC_{30}	2cee0c7500da619ce4ed0353600ed0d9
RC_{11}	fa0478a6de6f55724aaa9ec85c9d2d8a	RC_{31}	f0b1a5a196e90cab80bbbabc63a4a350
RC_{12}	dfb49f2b6b772a120efa4f2e29129fd4	RC_{32}	ae3db1025e962988ab0dde30938dca39
RC_{13}	1ea10344f449a23632d611aebb6a12ee	RC_{33}	17bb8f38d554a40b8814f3a82e75b442
RC_{14}	af0449884b0500845f9600c99ca8eca6	RC_{34}	34bb8a5b5f427fd7aeb6b779360a16f6
RC_{15}	21025ed89d199c4f78a2c7e327e593ec	RC_{35}	26f65241cbe5543843ce5918ffbaafde
RC_{16}	bf3aaaf8a759c9b7b9282ecd82d40173	RC_{36}	4ce99a54b9f3026aa2ca9cf7839ec978
RC_{17}	6260700d6186b01737f2efd910307d6b	RC_{37}	ae51a51a1bdff7be40c06e2822901235
RC_{18}	5aca45c22130044381c29153f6fc9ac6	RC_{38}	a0c1613cba7ed22bc173bc0f48a659cf
RC_{19}	9223973c226b68bb2caf92e836d1943a	RC_{39}	756acc03022882884ad6bdfde9c59da1

B Test Vectors for Haraka v2

Haraka-512 v2

```
Input:  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
        10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
        20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
        30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
Output: be 7f 72 3b 4e 80 a9 98 13 b2 92 28 7f 30 6f 62
        5a 6d 57 33 1c ae 5f 34 dd 92 77 b0 94 5b e2 aa
```

Haraka-256 v2

```
Input:  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
        10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
Output: 80 27 cc b8 79 49 77 4b 78 d0 54 5f b7 2b f7 0c
        69 5c 2a 09 23 cb d4 7b ba 11 59 ef bf 2b 2c 1c
```

C Active S-boxes

Table 10: Lower bound on the number of active S-boxes in a differential trail for the permutation and for the permutation when used in our mode for π_{512} ((a), (b), (c)) and for π_{256} ((d), (e), (f)). The cell color indicates the number of active S-boxes per total number of AES rounds (more transparent means fewer active).

(a) π_{512} DM-permutation

	m				
T	1	2	3	4	5
1	1	5	9	25	26
2	5	25	45	50	55
3	9	45	66	75	84
4	25	80	90	100	125
5	41	130	114	125	154
6	60	150	138	150	195
7	64	170	162	175	224

(b) π_{512} permutation used in DM-mode

	m				
T	1	2	3	4	5
1	0	3	7	17	25
2	3	17	37	46	53
3	7	37	58	71	82
4	17	72	82	96	123
5	33	128	106	121	152
6	52	142	130	146	193
7	60	162	154	171	222

(c) π_{512} permutation used in DM-mode leading to collision

	m				
T	1	2	3	4	5
1	0	9	13	17	25
2	12	34	37	46	58
3	18	76	60	71	91
4	32	93	84	96	128
5	39	134	108	121	161
6	52	159	132	146	198
7	60	198	156	171	231

(d) π_{256} permutation

	m				
T	1	2	3	4	5
1	1	5	9	25	26
2	5	25	40	50	55
3	9	35	59	75	84
4	25	60	80	100	125
5	34	80	101	125	153
6	45	100	122	150	190
7	52	110	143	175	221

(e) π_{256} permutation used in DM-mode

	m				
T	1	2	3	4	5
1	1	5	9	25	26
2	5	25	40	50	55
3	9	35	59	75	84
4	25	60	80	100	125
5	34	80	101	125	153
6	45	100	122	150	190
7	52	110	143	175	221

(f) π_{256} permutation used in DM-mode leading to collision

	m				
T	1	2	3	4	5
1	13	30	21	25	34
2	20	50	42	50	65
3	38	65	63	75	99
4	35	75	84	100	130
5	56	105	105	125	164
6	55	125	126	150	195
7	73	140	147	175	229

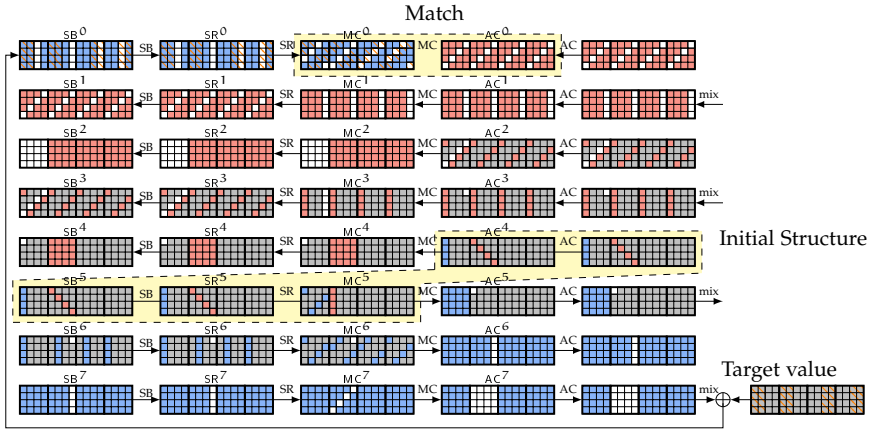


Figure 1: Meet-in-the-middle attack on 4 rounds of Haraka-512 v2. All □ are unknown, ▒ are constant, ■ neutral bytes backward, ■ neutral bytes forward and ▨ are the bytes truncated at the output.

D Meet-in-the-middle attack on Haraka-512 v2

For Haraka-512 v2 we can attack 4 rounds in the following way (see Figure 1):

1. Randomly select values for the constant bytes in AC^4 ▒.
2. For all 2^8 possible choices for $\mathsf{AC}^4_{*,0}$ which keep $\mathsf{MC}^4_{0,0}, \mathsf{MC}^4_{1,0}, \mathsf{MC}^4_{2,0}$ constant ■, compute forward to obtain the state in MC^0 and store the result in a table T.
3. For all 2^8 possible choices for $\mathsf{MC}^5_{*,4}$ which keep $\mathsf{AC}^5_{0,4}, \mathsf{AC}^5_{1,4}, \mathsf{AC}^5_{2,4}$ constant ■, compute backward to obtain the state in AC^0.
4. Check if there is an entry in T that matches with AC^0 through MixColumns. If so check whether the remaining bytes also match, otherwise repeat from step 2 (or step 1 if necessary).

Complexity. Computing the table T and 2^8 values for AC^0 costs 2^8 4-round Haraka-512 v2 evaluations and requires $2^8 \cdot 16$ bytes of memory. The success probability for the match is 2^{-32} for each state. Hence, on average $2^8 \cdot 2^8 \cdot 2^{-128} = 2^{-112}$ candidates will remain in Step 4. There are still 12 byte conditions which have to be satisfied in each state, which can be reduced to 4 byte conditions by using the fact that we can freely choose those bytes which are truncated at the output. Therefore, if we repeat step 1-4 2^{240} times we

expect to find $2^{240} \cdot 2^{-112} \cdot 2^{-4 \cdot 4 \cdot 8} = 1$ solution. The overall complexity is $2^{240} \cdot 2^8 = 2^{248}$ evaluations of Haraka-512 v2 to find a preimage.

E Considerations Regarding Modes of Operation and Linear Mixing

When designing the general constructions for the compression functions, we initially had three approaches in mind:

1. Davies-Meyer construction with a block cipher (referred to as dm),
2. Davies-Meyer construction with a permutation (referred to as dmperm), and
3. Sponge construction (referred to as sponge).

For the first construction, we used a state of two blocks initialized to zero. As part of the round function R_t, we would apply two parallel calls the AES as part of the **aes** operation. The actual bits of the message would be taken into the state over several rounds via a simple message expansion procedure. While the block cipher approach led to a small context size, the simplicity of the message expansion implied the possibility for the attacker to control differences injected even after many rounds, thus obtaining collisions by difference cancellation. While this can potentially be mitigated by a more complex message expansion, this would in turn lead to harder analysis and slower implementations.

In order to avoid the negative consequences on security from a too simple message expansion, and to performance from a too complex message expansion, we opted to abandon the block cipher-based approach of (1) in favor of a permutation-based approach. In particular, we load the full message into the state of the permutation from the beginning. As such, the state size for Haraka-512 v2 must be at least 64 bytes, while that of Haraka-256 v2 must be at least 32 bytes, or, equivalently $b = 4$ and $b = 2$ blocks, respectively. With this, we considered two general approaches, namely (2) and (3) above. Firstly, one approach is to use a Davies-Meyer construction where the message is loaded into the state which has the size of the domain in bits. This is the approach we landed on, and that described in Section 2 above. Finally, with a Sponge-based approach, one would choose the state size to be *larger* than the size of the domain. The state is initialized to some constant, e.g. all zeroes. The message is XORed into the most significant $|M|$ bits of the state, and a permutation is applied. The output is now taken as e.g. the most significant 256 bits in the case of both Haraka-512 v2 and Haraka-256 v2.

While the dm approach above was found to lead to significantly poorer security margins, in comparison to the dmperm and sponge approaches, we nevertheless implemented all three approaches in C.

For the sponge approach, we used a state consisting of 6 blocks, or, equivalently, 96 bytes. For dm, we used a state of 2 blocks, initialized to zero. The message expansion consisted of shuffling message bits and XORing them to other message bits, so, in other words, a simple linear expansion. In all cases, the permutation applied in each round had the form of **aes** (consisting of m rounds of the AES applied in parallel to each block of the state) followed by a linear mixing. Here, we focus on a fixed mixing layer (in particular using the **blend** mixing detailed below) while, in Section 5.2, we describe considerations regarding different approaches to the linear mixing.

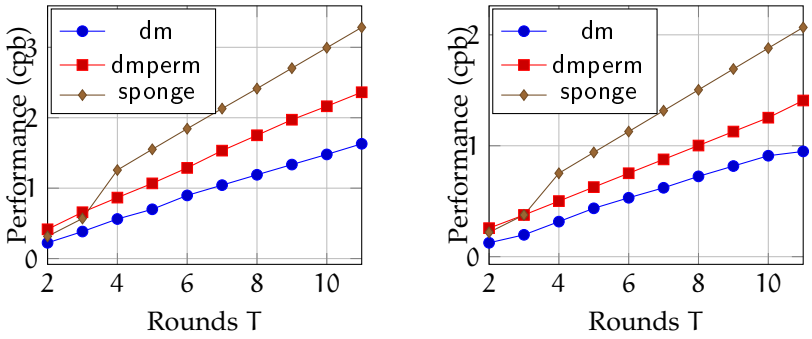
In our consideration here, the mixing layer is implemented by using the **blend** (or pblendw) instruction which is available in Intel CPUs supporting SSE 4.1. The **blend** instruction itself takes in two block operands and an 8-bit mask w. Let $y = \mathbf{blend}_w(a, b)$ be the blend operation on operands a and b using mask w. Then the ith least significant 16-bit word of y is determined as the corresponding word of either a or b, depending on the value of the ith bit of w. As such, **blend** gives us essentially a way to mix two blocks without permuting the byte positions. The mixing using **blend** is now defined as using $\mathbf{blend}_w$ on block i with block $i+1$ modulo the number of blocks of the state. Fixing $m = 2$, i.e. using two AES rounds per round, Figure 1 details the performance using the three general construction approaches dm, dmperm and sponge, described above. The numbers are taken as the minimum over choices of P in the range $P = 1, \ldots, 16$. Note, that the optimal choice for a particular value of P may not be constant across choices of the number of rounds T. Evidently, the dm approach has the best overall performance. The sponge approach is significantly slower than the dmperm approach when $T > 3$. To that end, and combined with the observation regarding the security properties of the dm approach, this led to the overall choice of the dmperm construction used for both Haraka-512 v2 and Haraka-256 v2.

For the linear mixing layer, we considered several possible approaches:

1. The $\mathbf{mix}_{512}$ and $\mathbf{mix}_{256}$ approaches described in Section 2, using the punpckhdq and punpckldq instructions;

2. The **blend** approach, as described above, using the pblendw instruction; and

3. Using a combination of a block-wise byte shuffle and XOR (denoted **shuffle-xor**) with the following state block, i.e. where block i updated with a byte shuffle and XORed with block $i+1$ modulo the number of



(a) Haswell (b) Skylake

Figure 1: Performance using $m = 2$ for each of the three general Haraka-512 v2 constructions considered

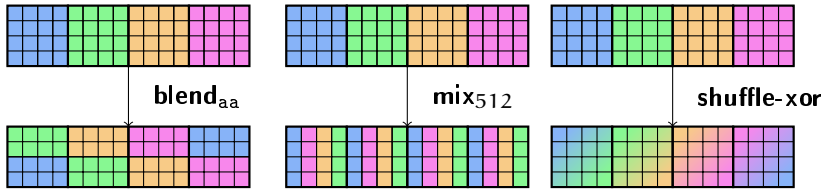
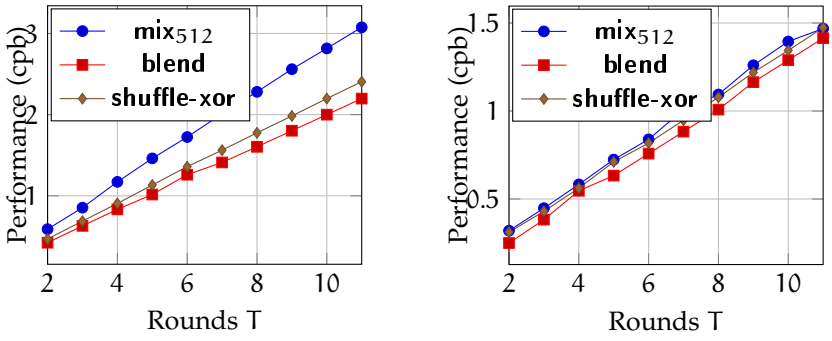


Figure 2: Effect of applying one round of the mixing layers on the state of π_{512}.

blocks, to obtain the updated block. This approach uses the pshufb and pxor instructions.



(a) Haswell (b) Skylake

Figure 3: Performance of Haraka-512 v2 using $m = 2$ for each of the three approaches to linear mixing considered

The effect of each of this operations applied to the state of π_{512} can be seen in Figure 2. On both the Haswell and Skylake microarchitectures, the instructions used for those three approaches all have a latency of one clock cycle, while the inverse throughput varies from e.g. 0.33 instructions/cycle for the XOR operation to 1 instruction/cycle for the punpckhdq and punpckldq instructions.

Figure 3 gives a performance comparison of the three approaches to the linear mixing layer. As shown, with the exception of the $\mathbf{mix}_{512}$ operation on Haswell, all other approaches have comparable performance for both Haswell and Skylake. Concludingly, it makes sense to choose the approach yielding the best security properties, namely the $\mathbf{mix}_{512}$ and $\mathbf{mix}_{256}$ operations.

The Skinny Family of Block Ciphers and its Low-Latency Variant Mantis

Publication Information

Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. "The SKINNY Family of Block Ciphers and its Low-Latency Variant MANTIS." In: *Advances in Cryptology - CRYPTO 2016*. 2016

Contribution

- Design and analysis of the linear layer for Skinny. Optimized software implementation and benchmarking.

Remarks

This is the full version of the original publication and has been slightly edited to fit the format.

The SKINNY Family of Block Ciphers and its Low-Latency Variant MANTIS
(Full Version)

Christof Beierle[1], Jérémy Jean[2], Stefan Kölbl[3], Gregor Leander[1], Amir Moradi[1], Thomas Peyrin[2], Yu Sasaki[4], Pascal Sasdrich[1], and Siang Meng Sim[2]

[1] Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
{Firstname.Lastname}@rub.de
[2] School of Physical and Mathematical Sciences
Nanyang Technological University, Singapore
Jean.Jeremy@gmail.com, Thomas.Peyrin@ntu.edu.sg, SSIM011@e.ntu.edu.sg
[3] DTU Compute, Technical University of Denmark, Denmark
stek@dtu.dk
[4] NTT Secure Platform Laboratories, Japan
Sasaki.Yu@lab.ntt.co.jp

Abstract. We present a new tweakable block cipher family SKINNY, whose goal is to compete with NSA recent design Simon in terms of hardware/software performances, while proving in addition much stronger security guarantees with regards to differential/linear attacks. In particular, unlike Simon, we are able to provide strong bounds for all versions, and not only in the single-key model, but also in the related-key or related-tweak model. SKINNY has flexible block/key/tweak sizes and can also benefit from very efficient threshold implementations for side-channel protection. Regarding performances, it outperforms all known ciphers for ASIC round-based implementations, while still reaching an extremely small area for serial implementations and a very good efficiency for software and micro-controllers implementations (SKINNY has the smallest total number of AND, OR, XOR gates used for encryption process).

Secondly, we present MANTIS, a dedicated variant of SKINNY for low-latency implementations, that constitutes a very efficient solution to the problem of designing a tweakable block cipher for

This article is the full version of the paper published in the proceedings of CRYPTO 2016, ©IACR 2016. Updated information on SKINNY will be made available via https://sites.google.com/site/skinnycipher/.

memory encryption. MANTIS basically reuses well understood, previously studied, known components. Yet, by putting those components together in a new fashion, we obtain a competitive cipher to PRINCE in latency and area, while being enhanced with a tweak input.

Keywords: lightweight encryption, low-latency, tweakable block cipher, MILP.

1 Introduction

Due to the increasing importance of pervasive computing, lightweight cryptography is currently a very active research domain in the symmetric-key cryptography community. In particular, we have recently seen the apparition of many (some might say too many) lightweight block ciphers, hash functions and stream ciphers. While the term *lightweight* is not strictly defined, it most often refers to a primitive that allows compact implementations, i.e. minimizing the area required by the implementation. While the focus on area is certainly valid with many applications, most of them require additional performance criteria to be taken into account. In particular, the throughput of the primitive represents an important dimension for many applications. Besides that, power (in particular for passive RFID tags) and energy (for battery-driven device) may be major aspects.

Moreover, the efficiency on different hardware technologies (ASIC, FPGA) needs to be taken into account, and even micro-controllers become a scenario of importance. Finally, as remarked in [4], software implementations should not be completely ignored for these lightweight primitives, as in many applications the tiny devices will communicate with servers handling thousands or millions of them. Thus, even so research started by focusing on chip area only, lightweight cryptography is indeed an inherent multidimensional problem.

Investigating the recent proposals in more detail, a major distinction is eye-catching and one can roughly split the proposals in two classes. The first class of ciphers uses very strong, but less efficient components (like the Sbox used in PRESENT [8] or LED [20], or the MDS diffusion matrix in LED or PICCOLO [38]). The second class of designs uses very efficient, but rather weak components (like the very small KATAN [13] or SIMON [3] round function).[1]

[1] Actually, this separation is not only valid for lightweight designs. It can well be extended to more classical ciphers or hash functions as well.

From a security viewpoint, the analysis of the members of the first class can be conducted much easily and it is usually possible to derive strong arguments for their security. However, while the second class strategy usually gives very competitive performance figures, it is much harder with state-of-the-art analysis techniques to obtain security guarantees even with regards to basic linear or differential cryptanalysis. In particular, when using very light round functions, bounds on the probabilities of linear or differential characteristics are usually both hard to obtain and not very strong. As a considerable fraction of the lightweight primitives proposed got quickly broken within a few months or years from their publication date, being able to give convincing security arguments turns out to be of major importance.

Of special interest, in this context, is the recent publication of the SIMON and SPECK family of block ciphers by the NSA [3]. Those ciphers brought a huge leap in terms of performances. As of today, these two primitives have an important efficiency advantage against all its competitors, in almost all implementation scenarios and platforms. However, even though SIMON or SPECK are quite elegant and seemingly well-crafted designs, these efficiency improvements came at an essential price. Echoing the above, since the ciphers have a very light round function, their security bounds regarding classical linear or differential cryptanalysis are not so impressive, quite difficult to obtain or even non-existent. For example, in [27] the authors provide differential/linear bounds for SIMON, but, as we will see, one needs a big proportion of the total number of rounds to guarantee its security according to its block size. Even worse, no bound is currently known in the related-key model for any version of SIMON and thus there is a risk that good related-key differential characteristics might exist for this family of ciphers (while some lightweight proposals such as LED [20], PICCOLO [38] or some versions of TWINE [42] do provide such a security guarantee). One should be further cautious as these designs come from a governmental agency which does not provide specific details on how the primitives were built. No cryptanalysis was ever provided by the designers. Instead, the important analysis work was been carried out by the research community in the last few years and one should note that so far SIMON or SPECK remain unbroken.

It is therefore a major challenge for academic research to design a cipher that can compete with SIMON's performances and additionally provides the essential strong security guarantees that SIMON is clearly lacking. We emphasize that this is both a research challenge and, in view of NSA's efforts to propose SIMON into an ISO standard, a challenge that has likely a practical impact.

Lightweight Tweakable Block Ciphers and Side-Channel Protected Implementations.

We note that tiny devices are more prone to be deployed into insecure environments and thus side-channel protected implementations of lightweight encryption primitives is a very important aspect that should be taken care of. One might even argue that instead of comparing performances of unprotected implementations of these lightweight primitives, one should instead compare protected variants (this is the recent trend followed by ciphers like ZORRO [18] or PICARO [35] and has actually already been taken into account long before by the cipher NOEKEON [17]). One extra protection against side-channel attacks can be the use of leakage resilient designs and notably through an extra tweak input of the cipher. Such tweakable block ciphers are rather rare, the only such candidate being Joltik-BC [23] or the internal cipher from SCREAM [44]. Coming up with a tweakable block cipher is indeed not an easy task as one must be extremely careful how to include this extra input that can be fully controlled by the attacker.

Low-Latency Implementations for Memory Encryption.

One very interesting field in the area of lightweight cryptography is memory encryption (see e.g. [21] for an extensive survey of memory encryption techniques). Memory encryption has been used in the literature to protect the memory used by a process domain against several types of attackers, including attackers capable of monitoring and even manipulating bus transactions. Examples of commercial uses do not abound, but there are at least two: IBM's SecureBlue++ [46] and Intel's SGX whose encryption and integrity mechanisms have been presented by Gueron at RWC 2016.[2] No documentation seems to be publicly available regarding the encryption used in IBM's solution, while Intel's encryption method requires additional data to be stored with each cache line. It is optimal in the context of encryption with memory overhead, but if the use case does not allow memory overhead then an entirely different approach is necessary.

With a focus on data confidentiality, a tweakable block cipher in ECB mode would then be the natural, straightforward solution. However, all generic methods to construct a tweakable block cipher from a block cipher suffer from an increased latency. Therefore, there is a clear need for lightweight tweakable block ciphers which do not require whitening value derivation, have a latency similar to the best non-tweakable block ciphers, and that can also be used in modes of operation that do not require memory expansion and offer beyond-birthday-bound security.

[2] The slides can be found here.

While being of great practical impact and need, it is actually very challenging to come up with such a block cipher. It should have three main characteristics. First, it must be executed within a single clock cycle and with a very low latency. Second, a tweak input is required, which in the case of memory encryption will be the memory address. Third, as one necessarily has to implement encryption and decryption, it is desirable to have a very low overhead when implementing decryption on top of encryption. The first and the third characteristics are already studied in the block cipher PRINCE [10]. However, the second point, i.e. having a tweak input, is not provided by PRINCE. It is not trivial to turn PRINCE into a tweakable block cipher, especially without increasing the number of rounds (and thereby latency) significantly.

Our Contributions.

Our contributions are twofold. First, we introduce a new lightweight family of block ciphers: SKINNY. Our goal here is to provide a competitor to SIMONin terms of hardware/software performances, while proving in addition much stronger security guarantees with regard to differential/linear attacks. Second, we present MANTIS, a dedicated variant of SKINNY that constitutes a very efficient solution to the aforementioned problem of designing a tweakable block cipher for memory encryption.

Regarding SKINNY, we have pushed further the recent trend of having a SPN cipher with locally non-optimal internal components: SKINNY is an SPN cipher that uses a compact Sbox, a new very sparse diffusion layer, and a new very light key schedule. Yet, by carefully choosing our components and how they interact, our construction manages to retain very strong security guarantees. For all the SKINNY versions, we are able to prove using mixed integer linear programming (MILP) very strong bounds with respect to differential/-linear attacks, not only in the single-key model, but also in the much more involved related-key model. Some versions of SKINNY have a very large key size compared to its block size and this theoretically renders the bounds search space huge. Therefore, the MILP methods we have devised to compute these bounds for a SKINNY-like construction can actually be considered a contribution by itself. As we will see later, compared to SIMON, in the single-key model SKINNY needs a much lower proportion of its total number of rounds to provide a sufficient bound on the best differential/linear characteristic. In the related-key model, the situation is even more at SKINNY's advantage as no such bound is known for any version of SIMONas of today.

With regard to performance, SKINNY reaches very small area with serial ASIC implementations, yet it is actually the very first block cipher that leads to better performances than SIMONfor round-based ASIC implementations, arguably the most important type of implementation since it provides a very

good throughput for a reasonably low area cost, in contrary to serial implementations that only minimizes area. We also exhibit ASIC threshold implementations of our SKINNY variants that compare for example very favourably to **aes**-128 threshold implementations. As explained above, this is an integral part of modern lightweight primitives.

Regarding software, our implementations outperform all lightweight ciphers, except SIMONwhich performs slightly faster in the situation where the key schedule is performed only once. However, as remarked in [4], it is more likely in practice that the key schedule has to be performed everytime, and since SKINNY has a very lightweight key schedule we expect the efficiency of SKINNY software implementations to be equivalent to that of SIMON. This shows that SKINNY would perfectly fit a scenario where a server communicate with many lightweight devices. These performances are not surprising, in particular for bit-sliced implementations, as we show that SKINNY uses a much smaller total number of AND/NOR/XOR gates compared to all known lightweight block ciphers. This indicates that SKINNY will be competitive for most platforms and scenarios. Micro-controllers are no exception, and we show that SKINNY performs extremely well on these architectures.

We further remark that the decryption process of SKINNY has almost exactly the same description as the encryption counterpart, thus minimizing the decryption overhead.

We finally note that similarly to SIMON, SKINNY very naturally encompasses 64- or 128-bit block versions and a wide range of key sizes. However, in addition, SKINNY provides a tweakable capability, which can be very useful not only for leakage resilient implementations, but also to be directly plugged into higher-level operating modes, such as SCT [34]. In order to provide this tweak feature, we have generalized the STK construction [22] to enable more compact implementations while maintaining a high provable security level.

The SKINNY specifications are given in Section 2. The rationale of our design as well as various theoretical security and efficiency comparisons are provided in Section 3. Finally, we conducted a complete security analysis in Section 4 and we exhibit our implementation results in Section 5.

Regarding MANTIS, we propose in Section 6 a low-latency tweakable block cipher that reuses some design principles of SKINNY.[3] It represents a very efficient solution to the aforementioned problem of designing a tweakable block cipher tailored for memory encryption.

The main challenge when designing such a cipher is that its latency is directly related to the number of rounds. Thus, it is crucial to find a design, i.e. a round function and a tweak-scheduling, that ensures security already with

[3]For the genesis of the cipher MANTIS, we acknowledge the contribution of Roberto Avanzi, as specified in Section 6.

a minimal number of rounds. Here, components of the recently proposed block ciphers PRINCE and MIDORI [1] turn out to be very beneficial.

The crucial step in the design of MANTIS was to find a suitable tweak-scheduling that would ensure a high number of active Sboxes not only in the single-key setting, but also in the setting where the attacker can control the difference in the tweak. Using, again, the MILP approach, we are able to demonstrate that a rather small number of rounds is already sufficient to ensure the resistance of MANTIS to differential (and linear) attacks in the related-tweak setting.

Besides the tweak-scheduling, we emphasize that MANTIS basically reuses well understood, previously studied, known components. It is mainly putting those components together in a new fashion, that allows MANTIS to be very competitive to PRINCE in latency and area, while being enhanced with a tweak. Thus, compared to the performance figures of PRINCE, we get the tweak almost for free, which is the key to solve the pressing problem of memory encryption.

2 Specifications of SKINNY

Notations and SKINNY Versions.

The lightweight block ciphers of the SKINNY family have 64-bit and 128-bit block versions and we denote n the block size. In both $n = 64$ and $n = 128$ versions, the internal state is viewed as a 4×4 square array of cells, where each cell is a nibble (in the $n = 64$ case) or a byte (in the $n = 128$ case). We denote $IS_{i,j}$ the cell of the internal state located at Row i and Column j (counting starting from 0). One can also view this 4×4 square array of cells as a vector of cells by concatenating the rows. Thus, we denote with a single subscript IS_i the cell of the internal state located at Position i in this vector (counting starting from 0) and we have that $IS_{i,j} = IS_{4 \cdot i+j}$.

SKINNY follows the TWEAKEY framework from [22] and thus takes a tweakey input instead of a key or a pair key/tweak. The user can then choose what part of this tweakey input will be key material and/or tweak material (classical block cipher view is to use the entire tweakey input as key material only). The family of lightweight block ciphers SKINNY have three main tweakey size versions: for a block size n, we propose versions with tweakey size $t = n$, $t = 2n$ and $t = 3n$ (versions with other tweakey sizes between n and $3n$ are naturally obtained from these main versions) and we denote $z = t/n$ the tweakey size to block size ratio. The tweakey state is also viewed as a collection of z 4×4 square arrays of cells of s bits each. We denote these arrays TK1 when $z = 1$, TK1 and TK2 when $z = 2$, and

finally TK1, TK2 and TK3 when $z = 3$. Moreover, we denote $TKz_{i,j}$ the cell of the tweakey state located at Row i and Column j of the z-th cell array. As for the internal state, we extend this notation to a vector view with a single subscript: $TK1_i$, $TK2_i$ and $TK3_i$. Moreover, we define the adversarial model **SK** (resp. **TK1**, **TK2** or **TK3**) where the attacker cannot (resp. can) introduce differences in the tweakey state.

Initialization.

The cipher receives a plaintext $m = m_0 \| m_1 \| \cdots \| m_{14} \| m_{15}$, where the m_i are s-bit cells, with $s = n/16$ (we have $s = 4$ for the 64-bit block SKINNY versions and $s = 8$ for the 128-bit block SKINNY versions). The initialization of the cipher's internal state is performed by simply setting $IS_i = m_i$ for $0 \leqslant i \leqslant 15$:

$$IS = \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

This is the initial value of the cipher internal state and note that the state is loaded row-wise rather than in the column-wise fashion we have come to expect from the **aes**; this is a more hardware-friendly choice, as pointed out in [29].

The cipher receives a tweakey input $tk = tk_0 \| tk_1 \| \cdots \| tk_{30} \| tk_{16z-1}$, where the tk_i are s-bit cells. The initialization of the cipher's tweakey state is performed by simply setting for $0 \leqslant i \leqslant 15$: $TK1_i = tk_i$ when $z = 1$, $TK1_i = tk_i$ and $TK2_i = tk_{16+i}$ when $z = 2$, and finally $TK1_i = tk_i$, $TK2_i = tk_{16+i}$ and $TK3_i = tk_{32+i}$ when $z = 3$. We note that the tweakey states are loaded row-wise.

Table 1: Number of rounds for SKINNY-n-t, with n-bit internal state and t-bit tweakey state.

	Tweakey size t		
Block size n	n	2n	3n
64	32 rounds	36 rounds	40 rounds
128	40 rounds	48 rounds	56 rounds

The Round Function.

One encryption round of SKINNY is composed of five operations in the following order: SubCells, AddConstants, AddRoundTweakey, ShiftRows and MixColumns (see illustration in Figure 1).

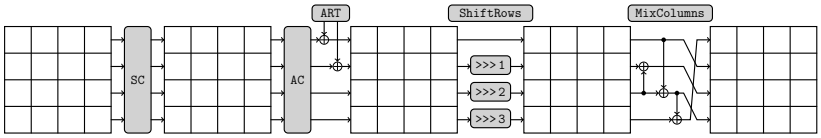


Figure 1: The SKINNY round function applies five different transformations: SubCells (SC), AddConstants (AC), AddRoundTweakey (ART), ShiftRows (SR) and MixColumns (MC).

The number r of rounds to perform during encryption depends on the block and tweakey sizes. The actual values are summarized in Table 1. Note that no whitening key is used in SKINNY. Thus, a part of the first and last round do not add any security. We motivate this choice in Section 3.

SubCells. A s-bit Sbox is applied to every cell of the cipher internal state. For $s = 4$, SKINNY cipher uses a Sbox $\mathcal{S}_4$ very close to the PICCOLO Sbox [38]. The action of this Sbox in hexadecimal notation is given by the following Table 2.

Table 2: 4-bit Sbox $\mathcal{S}_4$ used in SKINNY when $s = 4$.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$\mathcal{S}_4[x]$	c	6	9	0	1	a	2	b	3	8	5	d	4	e	7	f
$\mathcal{S}_4^{-1}[x]$	3	4	6	8	c	a	1	e	9	2	5	7	0	b	d	f

Note that $\mathcal{S}_4$ can also be described with four NOR and four XOR operations, as depicted in Figure 2. If x_0, x_1, x_2 and x_3 represent the four inputs bits of the Sbox (x_0 being the least significant bit), one simply applies the following transformation:

$$(x_3, x_2, x_1, x_0) \rightarrow (x_3, x_2, x_1, x_0 \oplus (\overline{x_3 \vee x_2})),$$

followed by a left shift bit rotation. This process is repeated four times, except for the last iteration where the bit rotation is omitted.

For the case $s = 8$, SKINNY uses an 8-bit Sbox $\mathcal{S}_8$ that is built in a similar manner as for the 4-bit Sbox $\mathcal{S}_4$ described above. The construction is

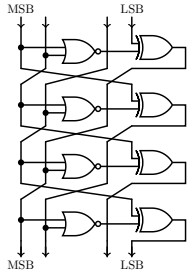


Figure 2: Construction of the Sbox $\mathcal{S}_4$.

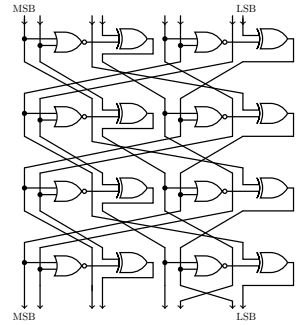


Figure 3: Construction of the Sbox $\mathcal{S}_8$.

simple and is depicted in Figure 3. If x_0, ..., x_7 represent the eight inputs bits of the Sbox (x_0 being the least significant bit), it basically applies the below transformation on the 8-bit state:

$$(x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0) \rightarrow (x_7, x_6, x_5, x_4 \oplus (\overline{x_7 \vee x_6}), x_3, x_2, x_1, x_0 \oplus (\overline{x_3 \vee x_2})),$$

followed by the bit permutation:

$$(x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0) \longrightarrow (x_2, x_1, x_7, x_6, x_4, x_0, x_3, x_5),$$

repeating this process four times, except for the last iteration where there is just a bit swap between x_1 and x_2. Besides, we provide in Section A the table of Sbox $\mathcal{S}_8$ and its inverse in hexadecimal notations.

`AddConstants`. A 6-bit affine LFSR, whose state is denoted $(rc_5, rc_4, rc_3, rc_2, rc_1, rc_0)$ (with rc_0 being the least significant bit), is used to generate round constants. Its update function is defined as:

$$(rc_5||rc_4||rc_3||rc_2||rc_1||rc_0) \rightarrow (rc_4||rc_3||rc_2||rc_1||rc_0||rc_5 \oplus rc_4 \oplus 1).$$

The six bits are initialized to zero, and updated *before* use in a given round. The bits from the LFSR are arranged into a 4×4 array (only the first column of the state is affected by the LFSR bits), depending on the size of internal state:

$$\begin{bmatrix} c_0 & 0 & 0 & 0 \\ c_1 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

with $c_2 = 0x2$ and

$(c_0, c_1) = (rc_3 \| rc_2 \| rc_1 \| rc_0,\ 0\|0\|rc_5\|rc_4)$ when $s = 4$
$(c_0, c_1) = (0\|0\|0\|0\|rc_3\|rc_2\|rc_1\|rc_0,\ 0\|0\|0\|0\|0\|0\|rc_5\|rc_4)$ when $s = 8$.

The round constants are combined with the state, respecting array positioning, using bitwise exclusive-or. The values of the constants for each round are given in the table below, encoded to byte values for each round, with rc_0 being the least significant bit.

Rounds	Constants
1 - 16	01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C,39,33,27,0E
17 - 32	1D,3A,35,2B,16,2C,18,30,21,02,05,0B,17,2E,1C,38
33 - 48	31,23,06,0D,1B,36,2D,1A,34,29,12,24,08,11,22,04
49 - 62	09,13,26,0C,19,32,25,0A,15,2A,14,28,10,20

`AddRoundTweakey`. The first and second rows of all tweakey arrays are extracted and bitwise exclusive-ored to the cipher internal state, respecting the array positioning. More formally, for $i = \{0, 1\}$ and $j = \{0, 1, 2, 3\}$, we have:

- $IS_{i,j} = IS_{i,j} \oplus TK1_{i,j}$ when $z = 1$,
- $IS_{i,j} = IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j}$ when $z = 2$,
- $IS_{i,j} = IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j} \oplus TK3_{i,j}$ when $z = 3$.

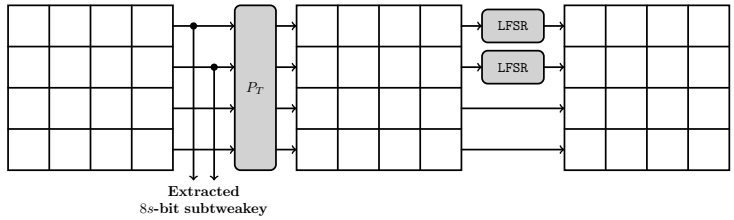


Figure 4: The tweakey schedule in SKINNY. Each tweakey word TK1, TK2 and TK3 (if any) follows a similar transformation update, except that no LFSR is applied to TK1.

Then, the tweakey arrays are updated as follows (this tweakey schedule is illustrated in Figure 4). First, a permutation P_T is applied on the cells

positions of all tweakey arrays: for all $0 \leqslant i \leqslant 15$, we set $TK1_i \leftarrow TK1_{P_T[i]}$ with

$$P_T = [9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7],$$

and similarly for TK2 when $z = 2$, and for TK2 and TK3 when $z = 3$. This corresponds to the following reordering of the matrix cells, where indices are taken row-wise:

$$(0, \ldots, 15) \xmapsto{P_T} (9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7)$$

Finally, every cell of the first and second rows of TK2 and TK3 (for the SKINNY versions where TK2 and TK3 are used) are individually updated with an LFSR. The LFSRs used are given in Table 3 (x_0 stands for the LSB of the cell).

Table 3: The LFSRs used in SKINNY to generate the round constants. The TK parameter gives the number of tweakey words in the cipher, and the s parameter gives the size of cell in bits.

TK	s	**LFSR**
TK2	4	$(x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_2\|\|x_1\|\|x_0\|\|x_3 \oplus x_2)$
	8	$(x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0\|\|x_7 \oplus x_5)$
TK3	4	$(x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_0 \oplus x_3\|\|x_3\|\|x_2\|\|x_1)$
	8	$(x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_0 \oplus x_6\|\|x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1)$

ShiftRows. As in **aes**, in this layer the rows of the cipher state cell array are rotated, but they are to the right. More precisely, the second, third, and fourth cell rows are rotated by 1, 2 and 3 positions to the right, respectively. In other words, a permutation P is applied on the cells positions of the cipher internal state cell array: for all $0 \leqslant i \leqslant 15$, we set $IS_i \leftarrow IS_{P[i]}$ with

$$P = [0, 1, 2, 3, 7, 4, 5, 6, 10, 11, 8, 9, 13, 14, 15, 12].$$

MixColumns. Each column of the cipher internal state array is multiplied by the following binary matrix $\mathbf{M}$:

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}.$$

The final value of the internal state array provides the ciphertext with cells being unpacked in the same way as the packing during initialization. Test vectors for SKINNY are provided in Section B. Note that decryption is very similar to encryption as all cipher components have very simple inverse (SubCells and MixColumns are based on a generalized Feistel structure, so their respective inverse is straightforward to deduce and can be implemented with the exact same number of operations).

Extending to Other Tweakey Sizes.

The three main versions of SKINNY have tweakey sizes $t = n$, $t = 2n$ and $t = 3n$, but one can easily extend this to any size[4] of tweakey $n \leqslant t \leqslant 3n$:

- for any tweakey size $n < t < 2n$, one simply uses exactly the $t = 2n$ version but the last $2n - t$ bits of the tweakey state are fixed to the zero value. Moreover, the corresponding cells in the tweakey state TK2 will not be updated throughout the rounds with the LFSR.
- for any tweakey size $2n < t < 3n$, one simply uses exactly the $t = 3n$ version but the last $3n - t$ bits of the tweakey state are fixed to the zero value. Moreover, the corresponding cells in the tweakey state TK3 will not be updated throughout the rounds with the LFSR.

We note that some of our 64-bit block SKINNY versions allow small key sizes (down to 64-bit). We emphasize that we propose these versions mainly for simplicity in the description of the SKINNY family of ciphers. Yet, as advised by the NIST [32], one should not to use key sizes that are smaller than 112 bits.

[4]For simplicity we do not include here tweakey sizes that are not a multiple of s bits. However, such cases can be trivially handled by generalizing the tweakey schedule description to the bit level.

Instantiating the Tweakey State with Key and Tweak Material.

Following the TWEAKEY framework [22], SKINNY takes as inputs a plaintext or a ciphertext and a tweakey value, which can be used in a flexible way by filling it with key and tweak material. Whatever the situation, the user must ensure that the key size is always at least as big as the block size.

In the classical setting where only key material is input, we use exactly the specifications of SKINNY described previously. However, when some tweak material is to be used in the tweakey state, we dedicate TK1 for this purpose and XOR a bit set to "1" every round to the second bit of the top cell of the third column (i.e. the second bit of $IS_{0,2}$). In other words, when there is some tweak material, we add an extra "1" in the constant matrix from AddConstants). Besides, in situations where the user might use different tweak sizes, we recommend to dedicate some cells of TK1 to encode the size of the tweak material, in order to ensure proper separation. Note that these are only recommendations, thus not strictly part of the specifications of SKINNY.

3 Rationale of SKINNY

Several design choices of SKINNY have been borrowed from existing ciphers, but most of our components are new, optimized for our goal: a cipher well suited for most lightweight applications. When designing SKINNY, one of our main criteria was to only add components which are vital for the security of the primitive, removing any unnecessary operation (hence the name of our proposal). We end up with the sound property that removing any component or using weaker version of a component from SKINNY would lead to a much weaker (or actually insecure) cipher. Therefore, the construction of SKINNY has been done through several iterations, trying to reach the exact spot where good performance meets strong security arguments. We detail in this section how we tried to follow this direction for each layer of the cipher.

We note that one could have chosen a slightly smaller Sbox or a slightly sparser diffusion layer, but our preliminary implementations showed that these options represent worse tradeoff overall. For example, one could imagine a very simple cipher iterating thousands of rounds composed of only a single non-linear boolean operation, an XOR and some bit wiring. However, such a cipher will lead to terrible performance regarding throughput, latency or energy consumption.

When designing a lightweight encryption scheme, several use cases must be taken in account. While area optimized implementations are important for some very constrained applications, throughput or throughput-over-area

optimized implementations are also very relevant. Actually, looking at recently introduced efficiency measurements [24], one can see that our designs choices are good for many types of implementations, which is exactly what makes a good general-purpose lightweight encryption scheme.

3.1 Estimating Area and Performances

In order to discuss the rationale of our design, we first quickly describe an estimation in Gate Equivalent (GE) of the ASIC area cost of several simple bit operations (for UMC 180nm 1.8 V [45]): a NOR/NAND gate costs 1 GE, a OR/AND gate costs 1.33 GE, a XOR/XNOR gate costs 2.67 GE and a NOT gate costs 0.67 GE. Finally, one memory bit can be estimated to 6 GE (scan flip-flop). Of course, these numbers depend on the library used, but it will give us at least some rough and easy evaluation of the design choices we will make.

Besides, even though many tradeoffs exist, we distinguish between a serial implementation, a round-based implementation and a low-latency implementation. In the latter, the entire ciphering process is performed in a single clock cycle, but the area cost is then quite important as all rounds need to be directly implemented. For a round-based implementation, an entire round of the cipher is performed in a single clock cycle, thus ending with the entire ciphering process being done in r cycles and with a moderate area cost (this tradeoff is usually a good candidate for energy efficiency). Finally, in a serial implementation, one reduces the datapath and thus the area to the minimum (usually a few bits, like the Sbox bit size), but the throughput is greatly reduced. The ultimate goal of a good lightweight encryption primitive is to use lightweight components, but also to ensure that these components are compact and efficient for all these tradeoffs. This is what Simon designers have managed to produce, but sacrificing a few security guarantees. SKINNY offers similar (sometimes even better) performances than Simon, while providing much stronger security arguments with regard to classical differential or linear cryptanalysis.

3.2 General Design and Components Rationale

A first and important decision was to choose between a Substitution Permutation Network (SPN), or a Feistel network. We started from a SPN construction as it is generally easier to provide stronger bounds on the number of active Sboxes. However, we note that there is a dual bit-sliced view of SKINNY that resembles some generalized Feistel network. Somehow, one can view the cipher as a primitive in between an SPN and an "AND-rotation-XOR" function like Simon. We try to get the best of both worlds by benefiting the nice

implementation tradeoffs of the latter, while organizing the state in an SPN view so that bounds on the number of active Sboxes can be easily obtained.

The absence of whitening key is justified by the reduction of the control logic: by always keeping the exact same round during the entire encryption process we avoid the control logic induced by having a last non-repeating layer at the end of the cipher. Besides, this simplifies the general description and implementation of the primitive. Obviously, having no whitening key means that a few operations of the cipher have no impact on the security. This is actually the case for both the beginning and the end of the ciphering process in SKINNY since the key addition is done in the middle of the round, with only half of the state being involved with this key addition every round.

A crucial feature of SKINNY is the easy generation of several block size or tweakey size versions, while keeping the general structure and most of the security analysis untouched. Going from the 64-bit block size versions to the 128-bit block size versions is simply done by using a 8-bit Sbox instead of a 4-bit Sbox, therefore keeping all the structural analysis identical. Using bigger tweakey material is done by following the STK construction [22], which allows automated analysis tools to still work even though the input space become very big (in short, the superposition trick makes the **TK2** and **TK3** analysis almost as time consuming as the normal and easy **TK1** case). Besides, unlike previous lightweight block ciphers, this complete analysis of the **TK2** and **TK3** cases allows us to dedicate a part of this tweakey material to be potentially some tweak input, therefore making SKINNY a flexible tweakable block cipher. Also, we directly obtain related-key security proofs using this general structure.

`SubCells.`

The choice of the Sbox is obviously a crucial decision in an SPN cipher and we have spent a lot of efforts on looking for the best possible candidate. For the 4-bit case, we have designed a tool that searches for the most compact candidate that provides some minimal security guarantees. Namely, with the bit operations cost estimations given previously, for all possible combinations of operations (NAND/NOR/XOR/XNOR) up to a certain limit cost, our tool checks if certain security criterion of the tested Sbox are fulfilled. More precisely, we have forced the maximal differential transition probability of the Sbox to be 2^{-2} and the maximal absolute linear bias to be 2^{-2}. When both criteria are satisfied, we have filtered our search for Sbox with high algebraic degree.

Our results is that the Sbox used in the PICCOLO block cipher [38] is close to be the best one: our 4-bit Sbox candidate $\mathcal{S}_4$ is essentially the PICCOLO Sbox with the last NOT gate at the end being removed (see Figure 2). We

believe this extra NOT gate was added by the PICCOLO designers to avoid fixed points (actually, if fixed points were to be removed at the Sbox level, the PICCOLO candidate would be the best choice), but in SKINNY the fixed points are handled with the use of constants to save some extra GE. Yet, omitting the last bit rotation layer removes already a lot of fixed points (the efficiency cost of this omission being null).

The Sbox $\mathcal{S}_4$ can therefore be implemented with only 4 NOR gates and 4 XOR gates, the rest being only bit wiring (basically free in hardware). According to our previously explained estimations, this should cost 14.68 GE, but as remarked in [38], some libraries provide special gates that further save area. Namely, in our library the 4-input AND-NOR and 4-input OR-NAND gates with two inputs inverted cost 2 GE and they can be used to directly compute a XOR or an XNOR. Thus, $\mathcal{S}_4$ can be implemented with only 12 GE. In comparison, the PRESENT Sbox [8] requires 3 AND, 1 OR and 11 XOR gates, which amounts to 27.32 GE (or 34.69 GE without the special 4-input gates).

All in all, our 4-bit Sbox $\mathcal{S}_4$ has the following security properties: maximal differential transition probability of 2^{-2}, maximal absolute linear bias of 2^{-2}, branching number 2, algebraic degree 3 and one fixed point $\mathcal{S}_4(\texttt{0xF}) = \texttt{0xF}$.

Regarding the 8-bit Sbox, the search space was too wide for our automated tool. Therefore, we instead considered a subclass of the entire search space: by reusing the general structure of $\mathcal{S}_4$, we have tested all possible Sboxes built by iterating several times a NOR/XOR combination and a bit permutation. Our search found that the maximal differential transition probability and maximal absolute linear bias of the Sboxes are larger than 2^{-2} when we have less than 8 iterations of the NOR/XOR combination and bit permutation. With 8 iterations of the NOR/XOR combination and bit permutation, we found Sboxes with desired maximal differential transition probability of 2^{-2} and maximal absolute linear bias of 2^{-2} with algebraic degree 6. However, the algebraic degree of the inverse Sboxes of all these candidates is 5 rather than 6. In addition, having 8 iterations may result in higher latency when we consider a serial hardware implementation. Therefore, we considered having 2 NOR/XOR combinations in every iteration and reduce the number of iteration from 8 to 4. As a result, we found several Sboxes with the desired maximal differential probability and absolute linear bias, while reaching algebraic degree 6 for both the Sbox and its inverse (thus better than the 8 iterations case). Although such Sbox candidates have 3 fixed points when we omit the last bit permutation layer like the 4-bit case, we can easily reduce the number of fixed points by introducing a different bit permutation from the intermediate bit permutations to the last layer without any additional cost.

With 2 NOR/XOR combinations and a bit permutation iterated 4 times, $\mathcal{S}_8$ can be implemented with only 8 NOR gates and 8 XOR gates (see Figure 3), the rest being only bit wiring (basically free in hardware). The total area cost should be 24 GE according to our previously explained estimations and using special 4-input AND-NOR and 4-input OR-NAND gates. In comparison, while ensuring a maximal differential transition probability (resp. maximum absolute linear bias) of 2^{-6} (resp. 2^{-4}), the **aes** Sbox requires 32 AND/OR gates and 83 XOR gates to be implemented, which amounts to 198 GE. Even recent lightweight 8-bit Sbox proposal [14] requires 12 AND/OR gates and 26 XOR gates, which amounts to 64 GE, for a maximal differential transition probability (resp. maximum linear bias) of 2^{-5} (resp. 2^{-2}), but their optimization goal was different from ours.

All in all, we believe our 8-bit Sbox candidate $\mathcal{S}_8$ provides a good tradeoff between security and area cost. It has maximal differential transition probability of 2^{-2}, maximal absolute linear bias of 2^{-2}, branching number 2, algebraic degree 6 and a single fixed point $\mathcal{S}_8(\texttt{0xFF}) = \texttt{0xFF}$ (for the Sbox we have chosen, swapping two bits in the last bit permutation was probably the simplest method to achieve only a single fixed point).

Note that both our Sboxes $\mathcal{S}_4$ and $\mathcal{S}_8$ have the interesting feature that their inverse is computed almost identically to the forward direction (as they are based on a generalized Feistel structure) and with exactly the same number of operations. Thus, our design reasoning also holds when considering the decryption process.

`AddConstants.`

The constants in SKINNY have several goals: differentiate the rounds (see Section 4.5), differentiate the columns and avoid symmetries, complicate subspace cryptanalysis (see Section 4.6) and attacks exploiting fixed points from the Sbox. In order to differentiate the rounds, we simply need a counter, and since the number of rounds of all SKINNY versions is smaller than 64, the most hardware friendly solution is to use a very cheap 6-bit affine LFSR (like in LED [20]) that requires only a single XNOR gate per update. The 6 bits are then dispatched to the two first rows of the first column (this will maximize the constants spread after the `ShiftRows` and `MixColumns`), which will already break the columns symmetry.

In order to avoid symmetries, fixed points and more generally subspaces to spread, we need to introduce different constants in several cells of the internal state. The round counter will already naturally have this goal, yet, in order to increase that effect, we have added a "1" bit to the third row, which is almost free in terms of implementation cost. This will ensure that

symmetries and subspaces are broken even more quickly, and in particular independently of the round counter.

`AddRoundTweakey`.

The tweakey schedule of SKINNY follows closely the STK construction from [22] (that allows to easily get bounds on the number of active Sboxes in the related-tweakey model). Yet, we have changed a few parts. Firstly, instead of using multiplications by 2 and 3 in a finite field, we have instead replaced these tweakey cells updates by cheap 4-bit or 8-bit LFSRs (depending on the size of the cell) to minimize the hardware cost. All our LFSRs require only a single XOR for the update, and we have checked that the differential cancellation behavior of these interconnected LFSRs is as required by the STK construction: for a given position, a single cancellation can only happen every 15 rounds for **TK2**, and same with two cancellations for **TK3**.

Another important generalization of the STK construction is the fact that every round we XOR only half of the internal cipher state with some subtweakey. The goal was clearly to optimize hardware performances of SKINNY, and it actually saves an important amount of XORs in a round-based implementation. The potential danger is that the bounds we obtain would dramatically drop because of this change. Yet, surprisingly, the bounds remained actually good and this was a good security/performance tradeoff to make. Another advantage is that we can now update the tweakey cells only before they are incorporated to the cipher internal state. Thus, half of tweakey cells only will be updated every round and the period of the cancellations naturally doubles: for a certain cell position, a single cancellation can only happen every 30 rounds for **TK2** and two cancellations can only happen every 30 rounds for **TK3**.

The tweakey permutation P_T has been chosen to maximize the bounds on the number of active Sboxes that we could obtain in the related-tweakey model (note that it has no impact in the single-key model). Besides, we have enforced for P_T the special property that all cells located in third and fourth rows are sent to the first and second rows, and vice-versa. Since only the first and second rows of the tweakey states are XORed to the internal state of the cipher, this ensures that both halves of the tweakey states will be equally mixed to the cipher internal state (otherwise, some tweakey bytes might be more involved in the ciphering process than others). Finally, the cells that will not be directly XORed to the cipher internal state can be left at the same relative position. On top of that, we only considered those variants of P_T that consist of a single cycle.

We note that since the cells of the first tweakey word TK1 are never updated, they can be directly hardwired to save some area if the situation allows.

ShiftRows and MixColumns.

Competing with SIMON's impressive hardware performance required choosing an extremely sparse diffusion layer for SKINNY, which was in direct contradiction with our original goal of obtaining good security bounds for our primitive. Note that since our Sboxes $\mathcal{S}_4$ and $\mathcal{S}_8$ have a branching number of two, we cannot use only a bit permutation layer as in the PRESENT block cipher: differential characteristics with only a single active Sbox per round would exist. After several design iterations, we came to the conclusion that binary matrices were the best choice. More surprisingly, while most block cipher designs are using very strong diffusion layers (like an MDS matrix), and even though a 4×4 binary matrices with branching number four exist, we preferred a much sparser candidate which we believe offers the best security/performance tradeoff (this can be measured in terms of Figure Of Adversarial Merit [24]).

Due to its strong sparseness, SKINNY binary diffusion matrix $\mathbf{M}$ has only a differential or linear branching number of two. This seems to be worrisome as it would again mean that differential characteristics with only a single active Sbox per round would exist (it would be the same for PRESENT block cipher if its Sbox did not have branching number three, which is the reason of the relatively high cost of the PRESENT Sbox). However, we designed $\mathbf{M}$ such that when a branching two differential transition occurs, the next round will likely lead to a much higher branching number. Looking at $\mathbf{M}$, the only way to meet branching two is to have an input difference in either the second or the fourth input only. This leads to an input difference in the first or third element for the next round, which then diffuses to many output elements. The differential characteristic with a single active Sbox per round is therefore impossible, and actually we will be able to prove at least 96 active Sboxes for 20 rounds. Thus, for the very cheap price of a differential branching two binary diffusion matrix, we are in fact getting a better security than expected when looking at the iteration of several rounds. The effect is the same with linear branching (for which we only need to look at the transpose of the inverse of $\mathbf{M}$, i.e. $(\mathbf{M}^{-1})^{\top}$).

We have considered all possibilites for $\mathbf{M}$ that can be implemented with at most three XOR operations and eventually kept the MixColumns matrices that, in combination with ShiftRows, guaranteed high diffusion and led to strong bounds on the minimal number of active Sboxes in the single-key model.

Note that another important criterion came into play regarding the choice of the diffusion layer of SKINNY: it is important that the key material impacts as fast as possible the cipher internal state. This is in particular a crucial point for SKINNY as only half of the state is mixed with some key material every round, and since there is no whitening keys. Besides, having a fast key diffusion will reduce the impact of meet-in-the-middle attacks. Once the two first rows of the state were arbitrarily chosen to receive the key material, given a certain subtweakey, we could check how many rounds were required (in both encryption and decryption directions) to ensure that the entire cipher state depends on this subtweakey. Our final choice of MixColumns is optimal: only a single round is required in both forward and backward directions to ensure this diffusion.

3.3 Comparing Differential Bounds

Our entire design has been crafted to allow good provable bounds on the minimal number of differential or linear active Sboxes, not only for the single-key model, but also in the related-key model (or more precisely the related-tweakey model in our case). We provide in Table 4 a comparison of our bounds with the best known proven bounds for other lightweight block ciphers at the same security level (all the ciphers in the table use 4-bit Sboxes with a maximal differential probability of 2^{-2}). We give in Section 4 more details on how the bounds of SKINNY were obtained.

First, we emphasize that most of the bounds we obtained for SKINNY are not tight, and we can hope for even higher minimal numbers of active Sboxes. This is not the case of LED or PRESENT for which the bounds are tight.

From the table, we can see that LED obtains better bounds for **SK**. Yet, the situation is inverted for **TK2**: due to a strong plateau effect in the **TK2** bounds of LED, it stays at 50 active Sboxes until Round 24, while SKINNY already reaches 72 active Sboxes at Round 24. Besides, LED performance will be quite bad compared to SKINNY, due to its strong MDS diffusion layer and strong Sbox.

Regarding PICCOLO, the bounds[5] are really similar to SKINNY for **SK** but worse for **TK2**. Yet, our round function is lighter (no use of a MDS layer), see Section 3.4.

No related-key bounds are known for MIDORI, PRESENT or TWINE. Besides, our **SK** bounds are better than PRESENT. Regarding MIDORI or TWINE in **SK**, while our bounds are slightly worse, we emphasize again that our round function is much lighter and thus will lead to much better performances.

[5]We estimate the number of active Sboxes for PICCOLO to $\lceil 4.5 \cdot N_f \rceil$, where N_f is the number of active F-functions taken from [38].

Table 4: Proved bounds on the minimal number of differential active Sboxes for SKINNY-64-128 and various lightweight 64-bit block 128-bit key ciphers. Model **SK** denotes the single-key scenario and model **TK2** denotes the related-tweakey scenario where differences can be inserted in both states TK1 and TK2.

Cipher	Model	Rounds															
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
SKINNY	**SK**	1	2	5	8	12	16	26	36	41	46	51	55	58	61	66	75
(36 rounds)	**TK2**	0	0	0	0	1	2	3	6	9	12	16	21	25	31	35	40
LED	**SK**	1	5	9	25	26	30	34	50	51	55	59	75	76	80	84	100
(48 rounds)	**TK2**	0	0	0	0	0	0	0	0	1	5	9	25	26	30	34	50
PICCOLO	**SK**	0	5	9	14	18	27	32	36	41	45	50	54	59	63	68	72
(31 rounds)	**TK2**	0	0	0	0	0	0	0	5	9	14	18	18	23	27	27	32
MIDORI	**SK**	1	3	7	16	23	30	35	38	41	50	57	62	67	72	75	84
(16 rounds)	**TK2**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PRESENT	**SK**	-	-	-	-	10	-	-	-	-	20	-	-	-	-	30	-
(31 rounds)	**TK2**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
TWINE	**SK**	0	1	2	3	4	6	8	11	14	18	22	24	27	30	32	-
(36 rounds)	**TK2**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Comparing differential bounds with SIMON is not as simple as with SPN ciphers. Yet, bounds on the best differential/linear characteristics for SIMON have been provided recently by [27].[6]

Assuming (very) pessimistically for SKINNY that a maximum differential transition probability of 2^{-2} is always possible for each active Sbox in the differential paths with the smallest number of active Sboxes, we can directly obtain easy bounds on the best differential/linear characteristics for SKINNY. We provide in Table 5 a comparison between SIMON and SKINNY versions for the proportion of total number of rounds needed to provide a sufficiently good differential characteristic probability bound according to the cipher block size. One can see that SKINNY needs a much smaller proportion of its total number of rounds compared to SIMON to ensure enough confidence with regards to simple differential/linear attacks. Actually the related-key ratios of SKINNY are even smaller than single-key ratios of SIMON (no related-key bounds are known as of today for SIMON).

Finally, in terms of diffusion, all versions of SKINNY achieve full diffusion after only 6 rounds (forwards or backwards), while SIMON versions with 64-bit block size requires 9 rounds, and even 13 rounds for SIMON versions with 128-bit block size [27] (**aes**-128 reaches full diffusion after 2 of its 10 rounds).

[6] Their article initially contained results only for the smallest versions of SIMON, but the authors provided us updated results for all versions of SIMON.

Table 5: Comparison between **aes**-128 and SIMON/SKINNY versions for the proportion of total number of rounds needed to provide a sufficiently good differential characteristic probability bound according to the cipher block size (i.e. $< 2^{-64}$ for 64-bit block size and $< 2^{-128}$ for 128-bit block size). Results for SIMON are updated results taken from [27].

Cipher	Single Key	Related Key
SKINNY-64-128	8/36 = **0.22**	15/36 = **0.42**
SIMON-64-128	19/44 = **0.43**	**no bound known**
SKINNY-128-128	15/40 = **0.37**	19/40 = **0.47**
SIMON-128-128	41/72 = **0.57**	**no bound known**
aes-128	4/10 = **0.40**	6/10 = **0.60**

Again, the diffusion comparison according to the total number of rounds is at SKINNY's advantage.

3.4 Comparing Theoretical Performance

After some minimal security guarantee, the second design goal of SKINNY was to minimize the total number of operations. We provide in Table 6 a comparison of the total number of operations per bit for SKINNY and for other lightweight block ciphers, as well as some quality grade regarding its ASIC area in a round-based implementation. We explain in details in Section C how these numbers have been computed.

One can see from the Table 6 that SIMON and SKINNY compare very favorably to other candidates, both in terms of number of operations and theoretical area grade for round-based implementations. This seems to confirm that when it comes to lightweight block ciphers, SIMON is probably the strongest competitor as of today. Besides, SKINNY has the best theoretical profile among all the candidates presented here, even better than SIMON for area. For speed efficiency, SKINNY outperforms SIMON when the key schedule is taken in account. This scenario is arguably the most important in practice: as remarked in [4], it is likely that lightweight devices will cipher very small messages and thus the back-end servers communicating with millions of devices will probably have to recompute the key schedule for every small message received.

In addition to its smaller key size, we note that KATAN-64-80 [13] theoretical area grade is slightly biased here as one round of this cipher is extremely light and such a round-based implementation would actually look more like

Table 6: Total number of operations and theoretical performance of SKINNY and various lightweight block ciphers. N denotes a NOR gate, A denotes a AND gate, X denotes a XOR gate.

Cipher	nb. of rds	gate cost (per bit per round)			nb. of op. w/o key sch.	nb. of op. w/ key sch.	round-based impl. area
		int. cipher	key sch.	total			
SKINNY-64-128	36	1 N 2.25 X	0.625 X	1 N 2.875 X	3.25 × 36 = 117	3.875 × 36 = **139.5**	1 + 2.67 × 2.875 = **8.68**
Simon-64/128	44	0.5 A 1.5 X	1.5 X	0.5 A 3.0 X	2 × 44 = 88	3.5 × 44 = **154**	0.67 + 2.67 × 3 = **8.68**
PRESENT-128	31	1 A 3.75 X	0.125 A 0.344 X	1.125 A 4.094 X	4.75 × 31 = 147.2	5.22 × 31 = **161.8**	1.5 + 2.67 × 4.094 = **12.43**
PICCOLO-128	31	1 N 4.25 X		1 N 4.25 X	5.25 × 31 = 162.75	5.25 × 31 = **162.75**	1 + 2.67 × 4.25 = **12.35**
KATAN-64-80	254	0.047 N 0.094 X	3 X	0.047 N 3.094 X	0.141 × 254 = 35.81	3.141 × 254 = **797.8**	0.19 + 2.67 × 3.094 = **8.45**
SKINNY-128-128	40	1 N 2.25 X		1 N 2.25 X	3.25 × 40 = 130	3.25 × 40 = **130**	1 + 2.67 × 2.25 = **7.01**
Simon-128/128	72	0.5 A 1.5 X	1 X	0.5 A 2.5 X	2 × 68 = 136	3 × 68 = **204**	0.67 + 2.67 × 2.5 = **7.34**
Noekeon-128	16	0.5 (A + N) 5.25 X	0.5 (A + N) 5.25 X	1 (A + N) 10.5 X	6.25 × 16 = 100	12.5 × 16 = **200**	2.33 + 2.67 × 10.5 = **30.36**
aes-128	10	4.25 A 16 X	1.06 A 3.5 X	5.31 A 19.5 X	20.25 × 10 = 202.5	24.81 × 10 = **248.1**	7.06 + 2.67 × 19.5 = **59.12**
SKINNY-128-256	48	1 N 2.25 X	0.56 X	1 N 2.81 X	3.25 × 48 = 156	3.81 × 48 = **183**	1 + 2.67 × 2.81 = **8.5**
Simon-128/256	72	0.5 A 1.5 X	1.5 X	0.5 A 3.0 X	2 × 72 = 144	3.5 × 72 = **252**	0.67 + 2.67 × 3 = **8.68**
aes-256	14	4.25 A 16 X	2.12 A 7 X	6.37 A 23 X	20.25 × 14 = 283.5	29.37 × 14 = **411.2**	8.47 + 2.67 × 23 = **69.88**

a serial implementation and will have a very low throughput (KATAN-64-80 has 254 rounds in total).

While Table 6 is only a rough indication of the efficiency of the various designs, we observe that the ratio between the Simon and SKINNY best software implementations, or the ratio between the smallest Simon and SKINNY round-based hardware implementations actually match the results from the table (See Section 5.3).

4 Security Analysis

In this section, we provide an in-depth analysis of the security of the SKINNY family of block ciphers. We emphasize that we do not claim any security in

the chosen-key or known-key model, but we *do* claim security in the related-key model. Moreover, we chose not to use any constant to differentiate between different block sizes or tweakey sizes versions of SKINNY, as we believe such a separation should be done at the protocol level, for example by deriving different keys (note that, if needed, this can easily be done by encoding these sizes and use them as fixed extra constant material every round).

4.1 Differential/Linear Cryptanalysis

In order to argue for the resistance of SKINNY against differential and linear attacks, we computed lower bounds on the minimal number of active Sboxes, both in the single-key and related-tweakey model. We recall that, in a differential (resp. linear) characteristic, an Sbox is called *active* if it contains a non-zero input difference (resp. input mask). In contrast to the single-key model, where the round tweakeys are constant and thus do not influence the activity pattern, an attacker is allowed to introduce differences (resp. masks) within the tweakey state in the related-tweakey model. For that, we considered the three cases of choosing input differences in **TK1** only, both **TK1** and **TK2**, and in all of the tweakey states **TK1**, **TK2** and **TK3**, respectively. Table 7 presents lower bounds on the number of differential active Sboxes for 1 up to 30 rounds. For computing these bounds, we generated a Mixed-Integer Linear Programming model following the approach explained in [31, 41]. We refer to Appendix D for more details on how these bounds were computed.

Table 7: Lowerbounds on the number of active Sboxes in SKINNY. Note that the bounds on the number of linear active Sboxes in the single-key model are also valid in the related-tweakey model. In case the MILP optimization was too long, we provide upper bounds between parentheses.

Model	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SK	1	2	5	8	12	16	26	36	41	46	51	55	58	61	66
TK1	0	0	1	2	3	6	10	13	16	23	32	38	41	45	49
TK2	0	0	0	0	1	2	3	6	9	12	16	21	25	31	35
TK3	0	0	0	0	0	0	1	2	3	6	10	13	16	19	24
SK Lin	1	2	5	8	13	19	25	32	38	43	48	52	55	58	64

Model	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
SK	75	82	88	92	96	102	108	(114)	(116)	(124)	(132)	(138)	(136)	(148)	(158)
TK1	54	59	62	66	70	75	79	83	85	88	95	102	(108)	(112)	(120)
TK2	40	43	47	52	57	59	64	67	72	75	82	85	88	92	96
TK3	27	31	35	43	45	48	51	55	58	60	65	72	77	81	85
SK Lin	70	76	80	85	90	96	102	107	(110)	(118)	(122)	(128)	(136)	(141)	(143)

For lower bounding the number of linear active Sboxes, we used the same approach. For that, we considered the inverse of the transposed linear transformation $\mathbf{M}^\top$. However, for the linear case, we only considered the single-key model. As it is described in [28], there is no cancellation of active Sboxes in linear characteristics. Thus, the bounds for **SK** give valid bounds also for the case where the attacker is allowed to not only control the message but also the tweakey input.

The above bounds are for single characteristic, thus it will be interesting to take a look at differentials and linear hulls. Being a rather complex task, we leave this as future work.

4.2 Meet-in-the-Middle Attacks

Meet-in-the-middle attacks have been applied to block ciphers e.g. [9, 15]. From its application to the SPN structure [37], the number of attacked rounds can be evaluated by considering the maximum length of three features, partial-matching, initial structure and splice-and-cut. This evaluation approach can be seen in the proposal of MIDORI.

Partial-matching cannot work if the number of rounds reaches full diffusion rounds in each of forward and backward directions. For SKINNY, full diffusion is achieved after 6 rounds forwards and backwards. Thus, partial-matching can work at most $(6-1)+(6-1)=10$ rounds. The length of the initial structure can also be bounded by the smaller number of full diffusion rounds in backwards and forwards and the maximum number that all tweakey cells impact to an Sbox. As a result, it works up to $6+2-1=7$ rounds for SKINNY. Splice-and-cut may extend the number of attack rounds up to the smaller number of full diffusion rounds minus one, which is $6-1=5$ in SKINNY. In the end, we conclude that meet-in-the-middle attack may work up to $10+7+5=22$ rounds. Consequently, the 32+ rounds of SKINNY provides a reasonable security margin.

Remarks on Biclique Cryptanalysis.

Biclique cryptanalysis improves the complexity of exhaustive search by computing only a part of encryption algorithm. The improved factor is often evaluated by the ratio of the number of Sboxes involved in the partial computation to all Sboxes in the cipher. The improved factor can be relatively big when the number of rounds in the cipher is small, which is not the case in SKINNY. We do not think improving exhaustive search by a small factor will turn into serious vulnerability in future. Therefore, SKINNY is not designed to resist biclique cryptanalysis with small improvement.

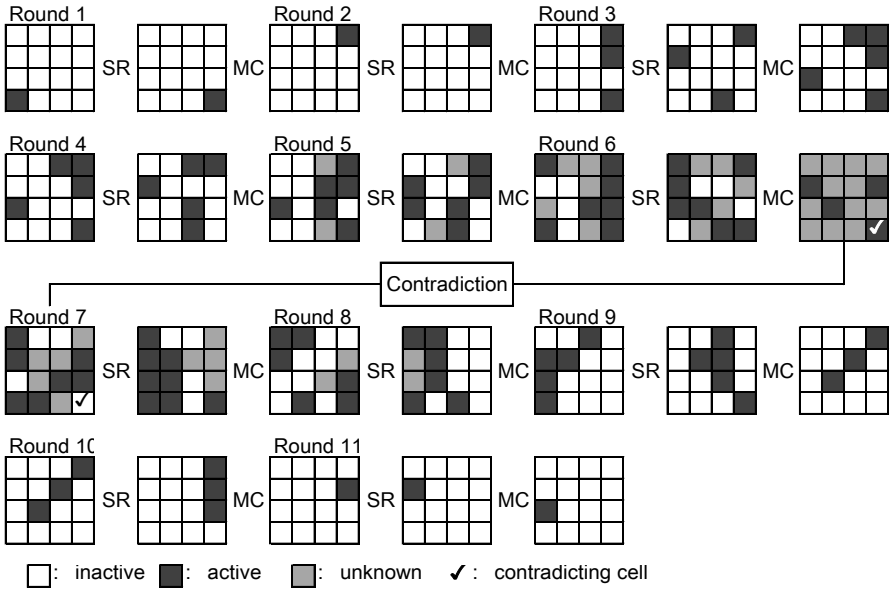


Figure 1: 11-round impossible differential characteristic. SR and MC stand for ShiftRows and MixColumns, respectively. SubCells, AddConstants and AddTweakey are omitted since they are not related to the impossible differential characteristic.

4.3 Impossible Differential Attacks

Impossible differential attack [5] finds two internal state differences Δ, Δ' such that Δ is never propagated to Δ'. The attacker then finds many pairs of plaintext/ciphertext and tweakey values leading to (Δ, Δ'). Those tweakey values are wrong values, thus tweakey space can be reduced.

We searched for impossible differential characteristics with the miss-in-the-middle technique. In short, 16 input truncated differentials and 16 output truncated differentials with single active cell are propagated with encryption function and decryption function, respectively, until no cell can be inactive or active with probability one. Then, we pick up the pair contradicting each other in the middle. Consequently, we found that the longest impossible differential characteristics reach 11 rounds and there are 16 such characteristics in total. An example of a 11-round impossible differential characteristic is as follows (also depicted in Figure 1):

$$(0,0,0,0,0,0,0,0,0,0,0,\Delta,0,0,0) \overset{12R}{\nrightarrow} (0,0,0,0,0,0,0,0,\Delta',0,0,0,0,0,0,0).$$

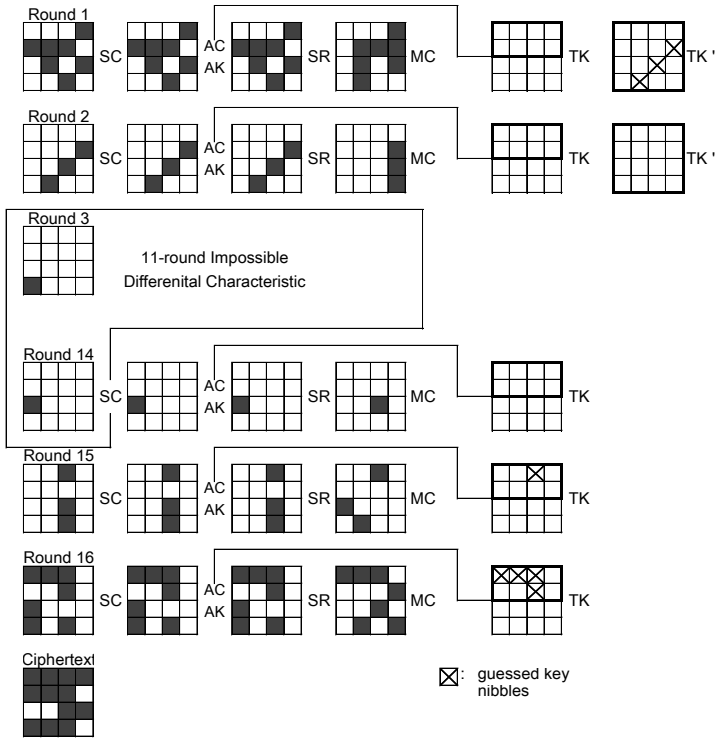


Figure 2: 16-round key recovery with impossible differential attack for SKINNY-64 with 64-bit tweakey and SKINNY-128 with 128-bit tweakey.

Several rounds can be appended before and after the 11-round impossible differential characteristic. The number of rounds appended depend on the key size. For example, when the block size and the key size are the same, two rounds and three rounds can be appended before and after the characteristic respectively, which makes 16-round key recovery. The plaintext difference becomes $(0,0,0,*,*,*,*,0,0,*,0,*,0,0,*,0)$ and the ciphertext difference becomes $(*,*,*,*,*,*,*,*,0,0,0,*,*,*,*,*,0)$, where $*$ denotes non-zero difference. The entire differential characteristic is illustrated in Figure 2.

The analysis is slightly different from standard SPN ciphers due to the lack of whitening key and unique order of the AddRoundTweakey (ARK) operation. For the first two rounds, ARK can be moved after the ShiftRows (SR) and MixColumns (MC) operations by applying the corresponding linear transformation to the tweakey value, which confirms that the first round acts as a keyless operation. Then, the analysis can start by regarding input difference to Round 2 as the plaintext difference and this is masked by the equivalent

tweakey for the first round. It also shows that the number of (equivalent) tweakey cells involved is 3 in the first three rounds and 5 in the last three rounds; hence, 8 cells in total.

The attacker constructs 2^x structures at the input to Round 2 and each structure consists of 2^{3c} values, where c is the cell size i.e. 4 bits for SKINNY-64 and 8 bits for SKINNY-128. In total, 2^{x+6c-1} pairs can be constructed from those 2^{x+3c} values. All the 2^{x+3c} values are inverted by one keyless round to obtain the corresponding original plaintexts and further queried the encryption oracle to obtain their corresponding ciphertexts. The attacker only picks up the pair which has 9 inactive cells after inverting the last MC operation. 2^{x-3c-1} pairs are expected to remain after the filtering. For each such pair, the attacker can generate all tweakey values for 8 cells leading to the impossible differential characteristic by guessing 5 internal state cells, which are 1-cell differences after MC in Round 3 and 4-cell differences before MC in Round 14 and Round 15. In the end, the attacker obtains $2^{x-3c-1+5c} = 2^{x+2c-1}$ wrong key suggestions for 8 tweakey cells, which makes the remaining tweakey space

$$2^{8c} \cdot (1 - 2^{-8c})^{2^{x+2c-1}} = 2^{8c} \cdot e^{-2^{x-6c-1}}.$$

When $c = 8$, we choose $x = 54.5$, which makes the remaining key space $2^{64} \cdot 2^{-65.3} < 1$. When $c = 4$, we choose $x = 29.5$, which makes the remaining key space $2^{32} \cdot 2^{-32.6} < 1$.

All in all, the data complexity amounts to 2^{x+3c} chosen plaintexts, and time and memory complexities are $\max\{2^{x+3c}, 2^{x+2c-1}\}$. Hence, data, time and memory complexities reach $2^{88.5}$ for SKINNY-128 with 128-bit key ($c = 8$) and $2^{41.5}$ for SKINNY-64 with 64-bit key ($c = 4$).

4.4 Integral Attacks

Integral attack [16, 26] prepares a set of plaintexts so that particular cells can contain all the values in the set and the other cells are fixed to a constant value. Then properties of the multiset of internal state values after encrypting several rounds are considered. In particular, we consider the following four properties.

All (A) : All values in the cell appear exactly the same number.

Balanced (B) : The sum of all values in the multiset is 0.

Constant (C) : The cell value is fixed through the multiset.

Unknown (U) : No particular property exists.

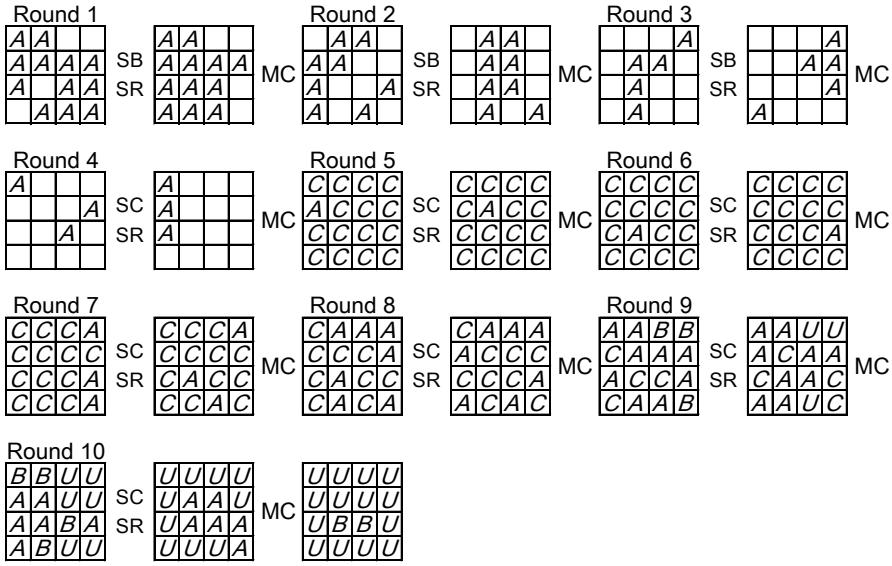


Figure 3: 10-round integral distinguisher. Rounds 5 to 10 show the property for 2^4 internal state values. Round 1 to 4 show which cells need to be active to extend it to higher-order integral property.

In order to find the maximum number of rounds preserving any non-trivial property, we follow an experimental approach. One active cell is set to the state, and those are processed by encryption algorithm until all cells become unknown. This is iterated many times by using different value of constant cells and tweakey. As a result, we found that an active cell in any of the third row will yield two cells satisfying the A property after seven rounds.

The property is then extended to higher-order by propagating the active cell in the backward direction. The property can be extended by 4 rounds in backwards by activating 12 cells. In the end, 10-round integral distinguishers can be constructed, which is illustrated in Figure 3.

Note that algebraic degree of the 4-bit Sbox is three (optimal) while algebraic degree of the 8-bit Sbox is six (not optimal), thus integral property of SKINNY-128 can be longer than SKINNY-64. We did the experiment for both versions, and found that the integral property was identical.

Key Recovery.

One can append 4 rounds after the 10-round integral distinguisher to make a 14-round key-recovery attack. The 4-round backward computation is depicted in Figure 4.

Then, the strategy proceeds as follows:

1. The attacker prepares 2^{12c} plaintexts to form the integral distinguisher. The attacker computes inverse MixColumns operation for each of the corresponding ciphertext, and takes parity of the 4-cell values necessary to proceed the backward computation. This reduces the remaining text size to 2^{8c}.

2. The attacker guesses 4 cells of the last tweakey and further computes inverse SubCells and inverse MixColumns. Again the attack takes the parity of 5-cell values after the inverse MixColumns. In the end, this step performs $2^{8c} \cdot 2^{4c} = 2^{12c}$ computations and obtain 2^{5c} data for each guess of 4 cells of tweakey. Note that guess-and-compress approach can be performed column-by-column, which can improve the complexity of this step smaller. However, we omit the very detailed optimization here.

3. Given 2^{5c} data, the attacker guesses 2 cells of tweakey in round 13 and computes back to 2 cells after the inverse MixColumns. The attack takes the parity of 2-cell values. 2^{2c} data are processed for $2^{4c+2c} = 2^{6c}$ tweakey guesses, which requires 2^{11c} computations and obtain 2^{2c} data for each guess of 6 cells of tweakey.

4. Given 2^{2c} data, 1 cell of tweakey in round 12 can be obtained from 4-cell guess for tweakey in round 14, which allows to compute back to the target cell in the output of the distinguisher. 2^{2c} data are processed for $2^{4c+2c} = 2^{6c}$ tweakey guesses, which requires 2^{8c} round function operations.

5. Computed results are tested if the Balanced (B) property is satisfied. The guessed 7-cell key candidates are reduced by a factor of 2^{-c}.

6. By iterating the analysis 5 times more, the 6-cell key candidates will be reduced to 1, and the other 10-cells can be guessed exhaustively.

Data complexity of this attack is 2^{12c} chosen plaintexts and memory access to deal with 2^{12c} ciphertext is the bottleneck of the complexity. The bottleneck of the memory complexity is also for the very first stage, which stores 2^{8c} state values after the first parity check.

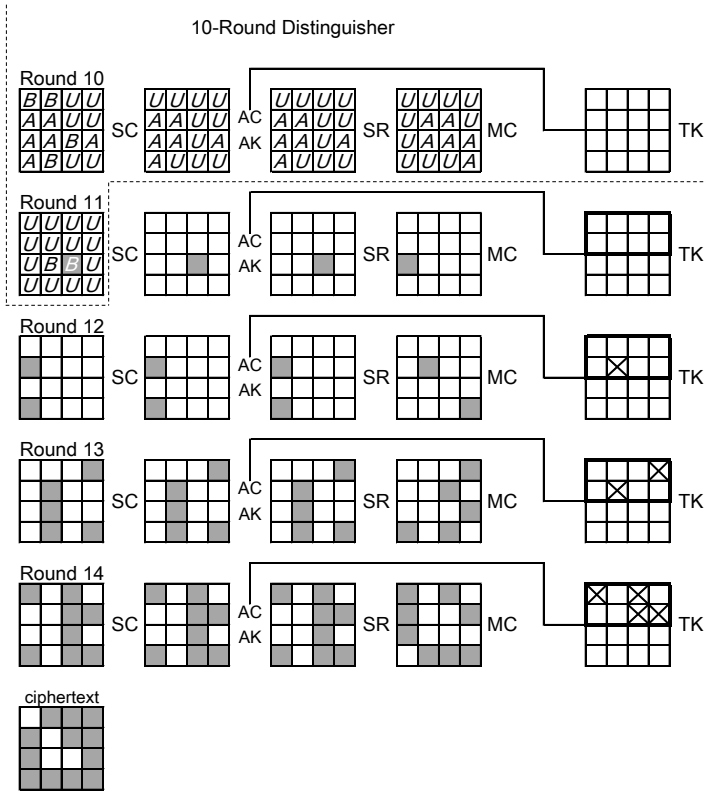


Figure 4: 16-round key recovery with integral attack for SKINNY-64 with 64-bit tweakey and SKINNY-128 with 128-bit tweakey. The cells that are involved during the last 2.5-round backward computation are colored in gray.

Remarks on the Division Property.

The division property was proposed by Todo [43] as a generalization of the integral property, which is in particular useful to precisely evaluate higher-order integral property. However, regarding application to SKINNY, experimental approach can reach more rounds. This is due to the very light round function, which allows relatively long integral property with a single active cell. In fact, an evaluation algorithm against generic SPN ciphers presented in [43, Algorithm 2] only leads to 6-round division property. Advanced evaluation algorithm needs to be developed to improve integral attack by division property.

A generalization of [43] described in [11] makes a link between the algebraic normal form (ANF) on an Sbox, and its resistance to the division prop-

erty. In particular, for a 4-bit mapping $(x_0, x_1, x_2, x_3) \rightarrow (y_0, y_1, y_2, y_3)$, they compute all the 16 possible products of y_i's terms and check which of the 16 possible products of x_i's appears in the resulting ANF. The result is shown graphically in Table 8. From an attacker point of view, the authors explain in [11] that the sparse lines can be used to launch an attack. While the resulting table in the case of SKINNY-64 seems sparser than the case of PRESENT for instance (which can be explained by the design strategy adopted), we are still confident that our proposals offer a strong security margin regarding this class of attacks.

Table 8: Division Property in the case of SKINNY-64 Sbox $(x_0, x_1, x_2, x_3) \rightarrow (y_0, y_1, y_2, y_3)$. For a given column α, the binary representation of $\alpha = \overline{\alpha_0\alpha_1\alpha_2\alpha_3}$ gives the ANF decomposition of the product of $\prod_{\alpha_i=1} y_i$ in terms of the products of x_j's. In particular, the four columns 1, 2, 4, 8 gives the ANF decomposition of the Sbox.

	0	1	2	4	8	3	5	9	6	a	c	7	b	d	e	f
0	X	X	X			X										
1		X	X	X	X	X					X					
2		X	X		X	X										
4			X		X	X		X								
8		X		X		X			X							
3		X		X		X					X					
5				X	X	X		X		X	X					
9					X	X			X		X					
6			X	X		X		X								
a					X	X	X		X							
c				X	X	X		X	X							
7				X	X	X		X	X						X	
b											X	X				
d									X		X		X			
e					X	X			X		X			X		
f																X

4.5 Slide Attacks

In SKINNY, the distinction between the rounds of the cipher is ensured by the AddConstants operation and thus the straightforward slide attacks cannot

be applied. However, the affine LFSR, which is the source of the distinction, has a state size of 6 bits. Hence, it occurs quite frequently that either $rc_5 \| rc_4$ or $rc_3 \| rc_2 \| rc_1 \| rc_0$ (the two constants cell values that depend on the round number) collides in different rounds, which could reduce the power of the round distinction.

We took into account all possible sliding numbers of rounds and deduced what is the difference in the constants that is obtained every time. As these constant differences might impact the best differential characteristic, we experimentally checked the lower bounds on the number of active Sboxes for all these constant differences by using MILP.

In the single-key setting, by allowing any starting round for each value of the slid pair, the lower bounds on the number of active Sboxes reach 36 after 11 rounds, and 41 after 12 rounds. All the pairs of starting rounds allowing these bounds are listed in Table 9. These bounds are not tight. Hence, they do not indicate the existence of exact differential characteristic matching the bounds. Moreover, in practice, attackers do not have any control on the input state of the middle rounds. From those reasons, we expect that slide attacks do not threat the security of SKINNY.

Table 9: Slid round numbers achieving the minimal lower bounds. The notation (a, b) means that the first and second values of the pair start from round a and round b, respectively. In this table, round numbers start from 0.

36 Sboxes for 11 rounds
$(12,31),(14,50),(16,30),(18,40),(1,25),(20,28),(21,38),(23,32),(26,47),(2,34),$ $(33,46),(36,39),(3,15),(42,49),(44,48),(4,10),(6,11),(7,17),(8,37),(9,29)$
41 Sboxes for 12 rounds
$(0,15),(10,45),(11,13),(12,37),(16,42),(17,18),(19,43),(21,33),(22,28),(24,29),$ $(25,35),(27,47),(2,44),(30,49),(34,48),(38,46),(41,50),(5,32),(7,40),(8,31)$

The similar attack can be evaluated in the related-key setting. Considering that the above discussion already assumes the very optimistic scenario for the attacker, i.e. the attacker can make reduced-round queries by forcing the oracle to start from any middle round of her choice, the impact to the real SKINNY seems very limited. We would leave this evaluation as future work.

4.6 Subspace Cryptanalysis

Invariant subspace cryptanalysis makes use of affine subspaces that are invariant under the round function. As the round key addition translates this

invariant subspace, ciphers exhibit weak keys when all round-keys are such that the affine subspace stays invariant including the key-addition. Therefore, those attacks are mainly an issue for block ciphers that use identical round keys. For SKINNY the non-trivial key-scheduling already provides a good protection against such attacks for a larger number of rounds. The main concern that remains are large-dimensional subspaces that propagate invariant through the Sbox. We checked that no such invariant subspaces exist. Moreover, for the 8-bit Sbox, we computed all affine subspaces of dimension larger than two that get mapped to (different) affine subspaces and checked if those can be chained to what could be coined a subspace characteristic (cf. [19] for a similar approach).

It turns out that those subspaces can be chained only for a very small number of rounds. Figure 5 shows as an example the affine spaces of dimension five. Thus to conclude, the non-trivial key-scheduling and the use of round-constants seem to sufficiently protect SKINNY against those attacks.

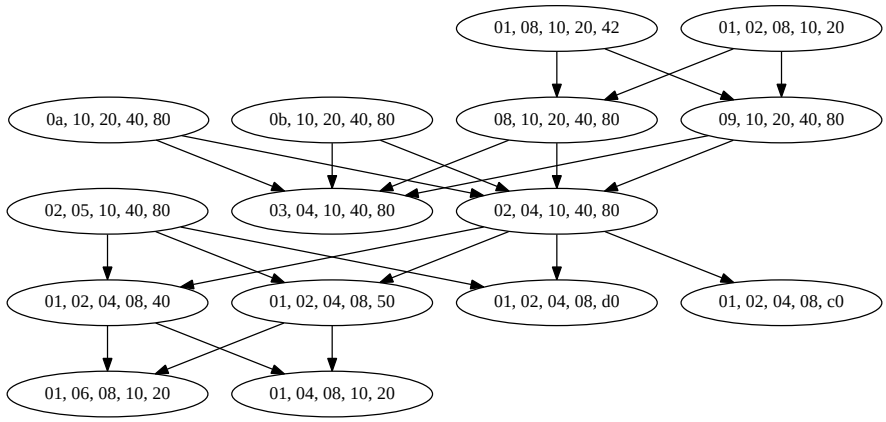


Figure 5: The graph showing all 5-dimensional affine spaces that gets mapped to (different) 5-dimensional spaces by applying the 8-bit Sbox of SKINNY-128. The nodes are the subspaces and the edges show which spaces are mapped to which spaces. The affine offset is ignored in this graph. The main point to make here is that the graph is actually a tree.

4.7 Algebraic Attacks

We argue why, not surprisingly, algebraic attacks do not threaten SKINNY. The Sbox $\mathcal{S}_4$ and $\mathcal{S}_8$ has algebraic degree $a = 3$ and $a = 6$ respectively. We can see from Table 4 that under the single-key scenario, for any consecutive 7-round differential characteristic of SKINNY, there are at least 26 active Sboxes. One

can easily check that for all SKINNY variants, we have $a \cdot 26 \cdot \lfloor \frac{r}{7} \rfloor \ggg n$, where r is the number of rounds and n is the block size. Moreover, $\mathcal{S}_4$ is described by $e = 21$ quadratic equations in the $v = 8$ input/output variables over $\mathrm{GF}(2)$. The entire system for a fixed-key SKINNY permutation therefore consists of $16 \cdot r \cdot e$ quadratic equations in $16 \cdot r \cdot v$ variables. For example, in the case of Skinny-64-64, there are 10752 quadratic equations in 4096 variables. In comparison, the entire system for a fixed-key **aes** permutation consists of 6400 equations in 2560 variables. While the applicability of algebraic attacks on **aes** remains unclear, those numbers tend to indicate that SKINNY offers a high level of protection.

5 Implementations, Performance and Comparison

5.1 ASIC Implementations

This section is dedicated to the description of the different hardware implementations of all variants of SKINNY. We used Synopsys DesignCompiler version A-2007.12-SP1 to synthesize the designs considering UMCL18G212T3 [45] standard cell library, which is based on the UMC L180 0.18µm 1P6M logic process with a typical voltage of 1.8 V. For the synthesis, we advised the compiler to keep the hierarchy and use a clock frequency of 100 KHz, which allows a fair comparison with the benchmark of other block ciphers reported in literature.

Round-Based Implementation.

In a first step, we designed round-based implementations for all SKINNY variants providing a good trade-off between performance and area. All implementations compute a single round of SKINNY within a clock cycle. Besides, our designs take advantage of dedicated scan flip-flops rather than using simple flip-flops and additional multiplexers placed in front in order to hold round states and keys. Note that this approach leads to savings of 1 GE per bit to be stored. In order to allow a better and fairer comparison, we provide both throughput at a maximally achievable frequency and throughput at a frequency of 100KHz.

Table 10 gives the area breakdown for round-based implementations of all SKINNY variants, while Table 11 compares our implementations with other round-based implementations of lightweight ciphers taken from the literature.

In particular, SKINNY-64-128 offers the smallest area footprint compared to other lightweight ciphers providing the same security level. Note, that even

Table 10: Area breakdown for round-based implementations of SKINNY-64 and SKINNY-128.

	64/64	64/128	64/192	128/128	128/256	128/384
	GE	GE	GE	GE	GE	GE
Key Schedule	384	789	1195	768	1557	2347
> Register	384	768	1152	768	1536	2304
> Logic	-	21	43	-	21	43
Round Function	839	907	988	1623	1755	1921
> Register	384	384	384	768	768	768
> Constant	42	42	42	42	42	42
> MixColumns	123	123	123	245	245	245
> Substitution	192	192	192	384	384	384
> Logic	98	166	247	184	316	482
Total	1223	1696	2183	2391	3312	4268

SIMON-64-128 implemented in a round-based fashion cannot compete with our design in terms of area although it has a smaller critical path, hence can be operated at higher frequencies and provides better throughput. However, comparing the throughput at a frequency of 100KHz, SKINNY provides better results since the number of rounds is substantially lower than for SIMON.

Using block sizes of 128 bits, SKINNY-128-128 is only slightly larger than SIMON-128-128, while SKINNY-128-256 again has a better area footprint. Besides, the throughput behaves in a similar manner as for SKINNY-64, since SIMON-128 still has a smaller critical path (due to less complex logic functions in terms of hardware gates). Still, it can be stated that SKINNY outperforms most existing lightweight ciphers, including SIMON, in terms of area and throughput considering hardware architectures in a round-based style.

Unrolled Implementation.

For the sake of completeness, we have investigated the area of SKINNY in a fully unrolled fashion. Unrolled implementations offer the best performance by computing a single encryption within one clock cycle. Therefore, all rounds are completely unrolled and the entire encryption or decryption function is implemented as combinatorial circuit at the disadvantage of increasing the critical path. However, this implementation can refrain from using registers to store intermediate values.

Table 11: Round-based implementations of SKINNY-64 and SKINNY-128.

	Area	Delay	Clock Cycles	Throughput @100KHz	Throughput @maximum	Ref.
	GE	ns	#	KBit/s	MBit/s	
SKINNY-64-64	1223	1.77	32	200.00	1130.00	**New**
SKINNY-64-128	1696	1.87	36	177.78	951.11	**New**
SKINNY-64-192	2183	2.02	40	160.00	792.00	**New**
SKINNY-128-128	2391	2.89	40	320.00	1107.20	**New**
SKINNY-128-256	3312	2.89	48	266.67	922.67	**New**
SKINNY-128-384	4268	2.89	56	228.57	790.86	**New**
SIMON-64-128	1751	1.60	46	145.45	870.00	[3]
SIMON-128-128	2342	1.60	70	188.24	1145.00	[3]
SIMON-128-256	3419	1.60	74	177.78	1081.00	[3]
LED-64-64	2695	-	32	198.90	-	[20]
LED-64-128	3036	-	48	133.00	-	[20]
PRESENT-64-128	1884	-	32	200.00	-	[8]
PICCOLO-64-128	1773[i]	-	33	193.94	-	[38]

[i] This number includes 576 GE for key storage that is not considered in the original work.

In Table 12, we list results of unrolled implementations for all SKINNY variants and compare it to appropriate results taken from the literature. Obviously, SKINNY cannot compete with PRINCE considering fully unrolled implementations while it still has better area results than LED, PRESENT and PICCOLO (at least for 64-bit block size and 128-bit keys). Unfortunately, the literature does not provide any numbers for latency and throughput (except for PRINCE), so we cannot compare our designs in these terms.

Serial Implementation.

As a common implementation fashion for lightweight ciphers, we have also considered byte-, nibble-, and bit-serial architectures to examine the performance of SKINNY.

Serial implementations have the smallest area footprint for hardware implementations by updating only a small number of bits per clock cycle. However, the throughput and performance of such implementations is decreased

Table 12: Unrolled implementations of SKINNY-64 and SKINNY-128.

	Area	Delay	Throughput		Ref.
			@100KHz	@maximum	
	GE	ns	KBit/s	MBit/s	
SKINNY-64-64	13340	44.74	6400.00	1430.49	**New**
SKINNY-64-128	17454	51.59	6400.00	1240.55	**New**
SKINNY-64-192	21588	57.56	6400.00	1111.88	**New**
SKINNY-128-128	32415	97.93	12800.00	1307.06	**New**
SKINNY-128-256	46014	119.57	12800.00	1070.50	**New**
SKINNY-128-384	61044	131.96	12800.00	1000.00	**New**
LED-64-128	111496	-	6400.00	-	[10]
PRESENT-64-128	56722	-	6400.00	-	[10]
PICCOLO-64-128	25668	-	6400.00	-	[10]
PRINCE	8512	13.00	6400.00	4923.08	[30]

significantly. Often, only a single instance of an Sbox is implemented and re-used to update the internal state of the round function in a serial fashion. Depending on the size of the Sbox, we call these implementations nibble-serial (4-bit Sbox) or byte-serial (8-bit Sbox), respectively (as an example, see Figure 1). Besides, we provide bit-serial implementations for SKINNY-64 and SKINNY-128 which update only a single bit of the round state per clock cycle. These implementations benefit from the iterative structure of both 4- and 8-bit Sboxes of SKINNY allowing to compute them bit by bit in 4 respectively 8 clock cycles.

In Table 13, we list results for nibble-serial implementations of all SKINNY-64 variants as well as results for byte-serial implementations of all SKINNY-128 variants. Obviously, our implementations cannot compete with Simon considering nibble-serial and byte-serial implementations while area and performance results still are comparable to results for LED, PRESENT and PICCOLO found in the literature.

Furthermore, we provide in Table 14 results for bit-serial implementations for all SKINNY variants. To the best of our knowledge, no bit-serial implementations are available for LED, PRESENT and PICCOLO so we only can compare our results to Simon. Still, Simon outperforms our implementations in terms of area and performance, but we would like to emphasize that (so far) the

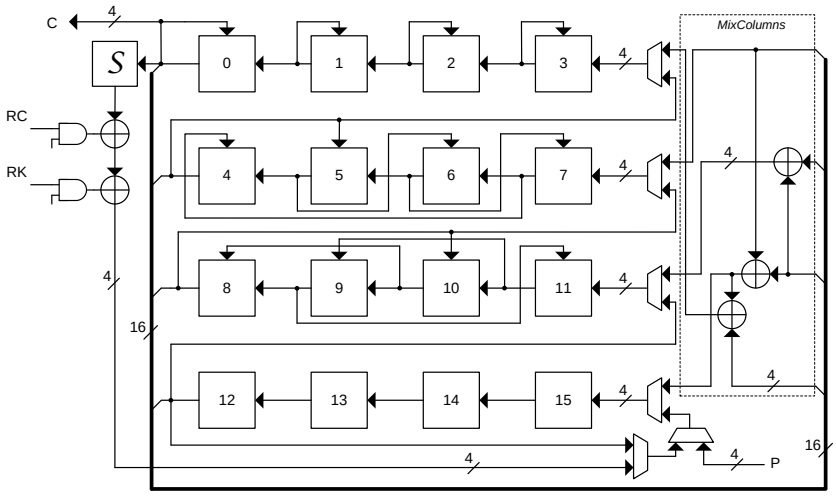


Figure 1: Hardware architecture of SKINNY-64 in a nibble-serial fashion

possibility of implementing an SPN cipher in a bit-serial way is an unique feature of SKINNY.

Threshold Implementation.

As a proper side-channel protection scheme for hardware platforms based on Boolean masking, we have realized first-order Threshold Implementation [33] of all variants of SKINNY. In short, thanks to the iterative architecture of the Sbox in SKINNY-128, its threshold implementation, compared to **aes** with the same Sbox size, is significantly smaller and faster, and does not need any fresh randomness.

We have designed the 3-share version of Threshold Implementations, where each single bit – in entire cipher internals – is represented with three shares, i.e., second-order Boolean masking. Due to the transparency of Boolean masking through linear operations, the 3-share representation of AddConstants, AddRoundTweakey, ShiftRows and MixColumns are easily achievable. However, the most challenging issue is to provide a uniform sharing of the non-linear functions, i.e., SubCells.

The Sbox $\mathcal{S}_4$ used in SKINNY-64 is a cubic bijection which – with respect to the categories given in [7] – belongs to the class $\mathcal{C}_{223}$, and can be decomposed to quadratic bijections with uniform sharing. However, considering the iterative construction of $\mathcal{S}_4$ (given in Section 2 and Figure 2), we decompose $\mathcal{S}_4$ into $\mathcal{G}_4$ and $\mathcal{F}_4$ in such a way that $\forall x, \mathcal{G}_4 \circ \mathcal{F}_4(\boldsymbol{x}) = \mathcal{S}_4(\boldsymbol{x})$. We define

$$\boldsymbol{y} :< y_3, y_2, y_1, y_0 >= \mathcal{F}_4(\boldsymbol{x} :< x_3, x_2, x_1, x_0 >)$$

Table 13: Serial implementations of SKINNY-64 (nibble) and SKINNY-128 (byte).

	Area	Delay	Clock Cycles	Throughput @100KHz	Throughput @maximum	Ref.
	GE	ns	#	KBit/s	MBit/s	
SKINNY-64-64	988	1.03	704	9.09	88.26	**New**
SKINNY-64-128	1399	0.95	788	8.12	85.49	**New**
SKINNY-64-192	1806	0.95	872	7.34	77.26	**New**
SKINNY-128-128	1840	1.03	872	14.68	142.51	**New**
SKINNY-128-256	2655	0.95	1040	12.31	129.55	**New**
SKINNY-128-384	3474	0.95	1208	10.60	111.54	**New**
Simon-64-128	1000	-	384	16.7	-	[3]
Simon-128-128	1317	-	560	22.9	-	[3]
Simon-128-256	1883	-	608	21.1	-	[3]
LED-64-64	966	-	1248	5.1	-	[20]
LED-64-128	1265	-	1872	3.4	-	[20]
PRESENT-64-128	1391	-	559	11.45	-	[8]
PICCOLO-64-128	758[i]	-	528	12.12	-	[38]

[i] This number includes 576 GE for key storage that is not considered in the original work.

as

$$y_0 = x_0 \oplus (\overline{x_2 \vee x_3})), \quad y_1 = x_1, \quad y_2 = x_2, \quad y_3 = x_3 \oplus (\overline{x_1 \vee x_2})),$$

and thanks to the iterative construction of $\mathcal{S}_4$, we can write $< y_3, y_2, y_1, y_0 >= \mathcal{G}_4(< x_3, x_2, x_1, x_0 >)$ as

$$< y_2, y_1, y_0, y_3 >= \mathcal{F}_4(< x_1, x_0, x_3, x_2 >),$$

which means that an input permutation and an output permutation over $\mathcal{F}_4$ realizes $\mathcal{G}_4$.

The transformation $\mathcal{F}_4$ is affine equivalent to the quadratic class $\mathcal{Q}_{294}$ [7], and its uniform sharing can be easily achieved by direct sharing. Let us represent $x_{i\in\{0,\ldots,3\}}$ with three shares (x_i^1, x_i^2, x_i^3), where $x_i = x_i^1 \oplus x_i^2 \oplus$

Table 14: Bit-serial implementations of SKINNY-64 and SKINNY-128.

	Area	Delay	Clock	Throughput		Ref.
			Cycles	@100KHz	@maximum	
	GE	ns	#	KBit/s	MBit/s	
SKINNY-64-64	839	1.03	2816	2.27	22.06	**New**
SKINNY-64-128	1172	1.06	3152	2.03	19.15	**New**
SKINNY-64-192	1505	1.00	3488	1.83	18.35	**New**
SKINNY-128-128	1481	1.05	6976	1.83	17.47	**New**
SKINNY-128-256	2125	0.89	8320	1.53	17.29	**New**
SKINNY-128-384	2761	0.89	9664	1.32	14.88	**New**
SIMON-64-128	958	-	1524	4.2	-	[3]
SIMON-128-128	1234	-	4414	2.9	-	[3]
SIMON-128-256	1782	-	4924	2.6	-	[3]

x_i^3. We define a component function $\boldsymbol{y} :< y_3, y_2, y_1, y_0 >= f_4(s, \boldsymbol{w} :< w_3, w_2, w_1, w_0 >, \boldsymbol{x} :< x_3, x_2, x_1, x_0 >)$ as

$$\begin{aligned} y_0 &= w_0 \oplus \overline{(s \vee (\overline{(\overline{x_2 \vee x_3}) \oplus (\overline{w_2 \vee x_3})}) \oplus (\overline{x_2 \vee w_3}))}, \\ y_1 &= w_1, \qquad y_2 = w_2, \\ y_3 &= w_3 \oplus \overline{(s \vee (\overline{(\overline{x_1 \vee x_2}) \oplus (\overline{w_1 \vee x_2})}) \oplus (\overline{x_1 \vee w_2}))}, \end{aligned}$$

which is made of only NOR and XOR gates. It is noteworthy that the extra input s controls the component function f_4 to pass the second input $\boldsymbol{w}$.

A uniform sharing of $\mathcal{F}_4$ over 3-share input $(\boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3)$ can be realized by three instances of f_4 as

$$\boldsymbol{y}^1 = f_4(s, \boldsymbol{x}^3, \boldsymbol{x}^2), \qquad \boldsymbol{y}^2 = f_4(s, \boldsymbol{x}^1, \boldsymbol{x}^3), \qquad \boldsymbol{y}^3 = f_4(s, \boldsymbol{x}^2, \boldsymbol{x}^1).$$

The same holds for $\mathcal{G}_4$, and by means of an input- and output permutation over f_4, we can realize its uniform sharing.

In Threshold Implementation of other ciphers, e.g., PRESENT [36], an extra register between the decomposed functions is required. However, in case of SKINNY, SubCells is performed prior to AddRoundTweakey, which allows us to place the uniform sharing of $\mathcal{F}_4$ between the state register (see Figure 2). Integrating the input s into the component functions f_4 turns $\mathcal{F}_4$ to operate as *pass through* which is required during MixColumns. Table 15 represents the

area overhead and performance of Threshold Implementation of all variants of SKINNY-64 based on a nibble-serial architecture.

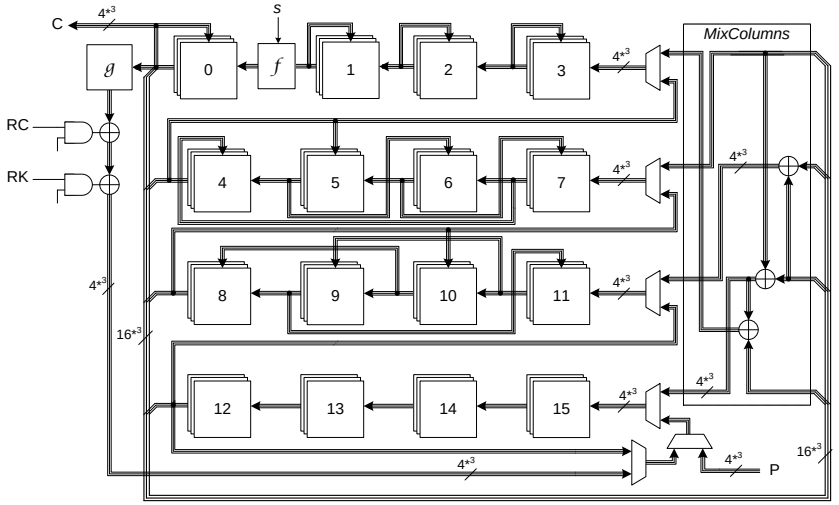


Figure 2: Hardware architecture of Threshold Implementation of SKINNY-64 in a nibble-serial fashion

We have applied the same concept on the SKINNY-128. In order to share the Sbox, we decompose $\mathcal{S}_8$ as $\mathcal{I}_8 \circ \mathcal{H}_8 \circ \mathcal{G}_8 \circ \mathcal{F}_8$, each of which is an 8-bit quadratic bijection. We define

$$\boldsymbol{y} :< y_7, y_6, y_5, y_4, y_3, y_2, y_1, y_0 >= \mathcal{F}_8(\boldsymbol{x} :< x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0 >)$$

as

$$\begin{aligned} y_0 &= x_0 \oplus (\overline{x_2 \vee x_3})), & y_1 &= x_1, & y_2 &= x_2, & y_3 &= x_3, \\ y_4 &= x_4 \oplus (\overline{x_6 \vee x_7})), & y_5 &= x_5, & y_6 &= x_6, & y_7 &= x_7. \end{aligned}$$

Other bijections are also defined over $\mathcal{F}_8$ as follows:

$$\begin{aligned} \mathcal{G}_8 : &< y_2, y_1, y_7, y_6, y_4, y_0, y_3, y_5 >= \mathcal{F}_8(< x_2, x_1, x_7, x_6, x_4, x_0, x_3, x_5 >) \\ \mathcal{H}_8 : &< y_0, y_3, y_2, y_1, y_6, y_5, y_4, y_7 >= \mathcal{F}_8(< x_0, x_3, x_2, x_1, x_6, x_5, x_4, x_7 >) \\ \mathcal{I}_8 : &< y_7, y_6, y_5, y_4, y_3, y_1, y_2, y_0 >= \mathcal{F}_8(< x_5, x_4, x_0, x_3, x_1, x_7, x_6, x_2 >) \end{aligned}$$

For a uniform sharing of each of these 8-bit bijections, we define a component function $\boldsymbol{y} :< y_7, y_6, y_5, y_4, y_3, y_2, y_1, y_0 >= f_8(s, \boldsymbol{w}, \boldsymbol{x})$ as

$$
\begin{aligned}
y_0 &= w_0 \oplus \overline{(s \vee \overline{((\overline{x_2 \vee x_3}) \oplus (\overline{w_2 \vee x_3}))} \oplus (\overline{x_2 \vee w_3}))},\\
y_1 &= w_1, \qquad y_2 = w_2, \qquad y_3 = w_3,\\
y_4 &= w_4 \oplus \overline{(s \vee \overline{((\overline{x_6 \vee x_7}) \oplus (\overline{w_6 \vee x_7}))} \oplus (\overline{x_6 \vee w_7}))},\\
y_5 &= w_5, \qquad y_6 = w_6, \qquad y_7 = w_7.
\end{aligned}
$$

Similar to f_4, the s input has been integrated in order to control the f_8 function to pass through the $\boldsymbol{w}$ input. Following the same concept as explained above, the 3-share input $(\boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3)$ can be given to three instances of f_8 to derive a 3-share uniform output of $\mathcal{F}_8$. Therefore, uniform sharing of all aforementioned 8-bit bijections can be achieved by input- and output permutations over the uniform sharing of $\mathcal{F}_8$. Further, we can place these shared functions between the state register in a serial implementation which avoids instantiating extra registers between the decomposed functions (see Figure 3).

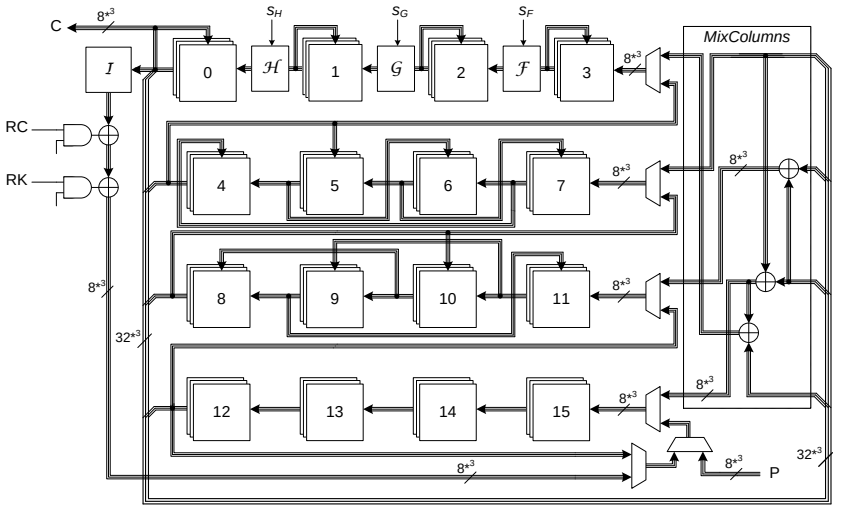


Figure 3: Hardware architecture of Threshold Implementation of SKINNY-128 in a byte-serial fashion

It should be noted that the above explained constructions allow extremely efficient Threshold Implementations since

- With a few NOR gates the functions are converted to pass through (required for MixColumns).

- The Sbox is decomposed to smaller functions with shorter critical path leading to designs with significantly high clock frequencies (see Table 15).
- Extra registers are avoided compared to e.g., [36].
- Our constructions do not need any fresh randomness during the entire operations of the cipher since we could provide the uniform sharing of the Sboxes compared to e.g., [6, 29]. Only the input (plaintext) should be masked using two random masks, each with the same length as the input.

In our Threshold Implementations – similar to many other Threshold Implementations reported in the literature [6, 29, 36] – only the state is masked, not the key registers, which is adequate to provide first-order security.

Table 15: Threshold implementations of SKINNY-64 (nibble-serial) and SKINNY-128 (byte-serial).

	Area	Delay	Clock	Throughput		Fresh	Ref.
			Cycles	@100KHz	@maximum	Rand.	
	GE	ns	#	KBit/s	MBit/s	bits	
SKINNY-64-64	1966	0.95	704	9.09	95.69	0	**New**
SKINNY-64-128	2372	1.00	788	8.12	81.22	0	**New**
SKINNY-64-192	2783	1.00	872	7.34	73.39	0	**New**
SKINNY-128-128	3780	1.63	872	14.68	90.05	0	**New**
SKINNY-128-256	4713	1.50	1040	12.31	82.05	0	**New**
SKINNY-128-384	5434	1.54	1208	10.60	68.80	0	**New**
AES-128	11114	-	266	48.12	-	7680	[29]
AES-128	8119	-	246	52.03	-	5120	[6]

5.2 FPGA Implementations

Today, FPGAs are used more and more for high-performance applications, even in the field of security and cryptographic applications. Since there are a wealth of different FPGA vendors available, we decided to implement our designs on Virtex-7 FPGAs provided by the market leader Xilinx. In this section, we provide detailed results of FPGA-tailored solutions for high-performance

implementations of SKINNY. Note, that it is almost a natural choice to implement high-throughput architectures on FPGAs using pipelining techniques since logic resources always come in conjunction with succeeding flip-flops. This allows to efficiently pipeline all computations at nearly no area overhead (in terms of occupied slices of the FPGA device) while keeping the critical path of the design at a minimum. Hence, the maximum frequency and finally the throughput of the design can be increased.

A brief summary of implementation results for high-performance architectures on FPGAs for both, SKINNY-64 and SKINNY-128, can be found in Table 16 providing details for used resources and achieved performance results. Note, however, that a fair comparison to existing work is rather difficult since most reference implementations found in the literature either do not target fully pipelined and unrolled implementations or just provide results for older FPGA technologies using 4-input LUTs instead of 6-input LUTs found in modern devices. Still, we would like to highlight the performance figures of all of our SKINNY implementations for FPGAs allowing to implement high-performance architectures at a minimum of resource consumption.

Table 16: High-throughput implementations of SKINNY-64 and SKINNY-128. Results are obtained after place-and-route for Virtex-7 XC7VX330T.

	Logic	**Memory**	**Frequency**	**T'put**	**Device**	**Ref.**
	LUT	FF	MHz	GBit/s	Xilinx	
SKINNY-64-64	3101	4000	403.88	25.85	Virtex-7	**New**
SKINNY-64-128	4247	6720	402.41	25.75	Virtex-7	**New**
SKINNY-64-192	6330	9952	400.48	25.63	Virtex-7	**New**
SKINNY-128-128	13389	10048	320.10	40.97	Virtex-7	**New**
SKINNY-128-256	17037	18048	355.62	45.52	Virtex-7	**New**
SKINNY-128-384	21966	28096	356.51	45.63	Virtex-7	**New**
ICEBERG-64-128	13616	-	297.00	19.01	Virtex-II	[39]
MISTY1-64-128	10920	8480	140.00	8.96	Virtex1000	[40]
KHAZAD-64-128	11072	9600	123.00	7.87	Virtex1000	[40]

5.3 Software Implementations

In this section, we detail how the ciphers in the SKINNY family can be implemented in software. More precisely, we consider four of the latest Intel processors using SIMD instruction sets to perform efficient parallel computations of several input blocks. We give in particular the performance figures for a bit-sliced implementations of SKINNY.

Notes on Previous Benchmarks and Comparisons.

In most of the previous designs proposed in academic publications, the designers give the full cost of encryption, including the costs to convert the data to the required form, the actual encryption, and possibly the expansion of the master key. This gives a broad overview of how well the cipher would behave in a more specific context, especially for bit-sliced implementation where packing and unpacking of the data can represent a non-negligible proportion of the encryption process.

In comparison, the SIMONimplementations from [2] do not include neither the cost of key expansion nor the cost of packing/unpacking the data, which prevents any meaningful comparison with the other lightweight ciphers having the same level of security (the argument to drop these costs relies on a strong restriction on the way the cipher implementations can be used).

In the following, we perform an evaluation of our proposals using four different recent high-speed platforms (exact setting given in Table 17) at different rates of parallelization. While we count the costs for packing and unpacking the data, we chose to benchmark encryption given pre-expanded subkeys. The motivation is twofold: first, many modes of operation make this assumption practical and second, the key schedules of our proposals are light and would not induce big differences in the results. At the end of this section, we also give a comparison of the speed SKINNY-64-128 can achieve in the case where the data packing is not needed (i.e. in the highly parallel counter mode considered in [3]).

Overall our benchmarking results show that the performance roughly follows what one would expect from Table 6. There are scenarios in practice for which the costs of the key schedule play a non-negligible role as pointed out in [4] and we expect the lower costs of the SKINNY key schedule to provide a good performance.

Bit-Sliced Implementations of SKINNY.

Since the design of SKINNY has been made with hardware implementations in mind, the conversion to bit-sliced implementations seems natural. In the

Table 17: Machine used to benchmark the software implementations (Turbo Boost disabled).

Name	Processor	Launch date	Linux kernel	gcc version
Westmere	X5650	Q1 2010	3.13.0-34	4.8.2
Ivy Bridge	i5-3470	Q2 2012	3.11.0-12	4.8.1
Haswell	i7-4770S	Q2 2013	4.4.0-22	5.3.1
Skylake	i7-6700	Q3 2015	4.2.3-040203	5.2.1

following, we target different sets of instructions, namely SSE4 and AVX2, which provide shuffling instructions on byte level, as well as several wide 128-bit resp. 256-bit registers, commonly referred as XMM or YMM registers. From our perspective, the main differences between SSE4 and AVX2 are the width of the available registers and the possibility to use 3-operand instructions.

In the Table 18, we give the detailed performance figures of our implementations in the case of SKINNY-64 and compare it with other ciphers. Note that these implementations take into account all data transformations which are required. The bit-sliced implementations for SIMONprocessing 32 resp. 64 blocks have been provided by the designers to allow us a fair comparison in the same setting.

Table 18: Bit-sliced implementations of SKINNY-64, SKINNY-128 and other 64-bit block lightweight ciphers. Performances are given in cycles per byte, with pre-expanded subkeys. For SKINNY-64 and SIMONwe encrypted 2000 64-bit blocks to obtain the results. Cells with dashes (-) represent non-existing implementations to date.

	Haswell			Skylake			Ref.
Parallelization ρ	16	32	64	16	32	64	
SKINNY-128-128	-	-	4.32	-	-	3.96	**New**
SKINNY-64-128	-	-	3.05	-	-	2.78	**New**
SIMON-64-128	-	3.42	1.93	-	3.29	1.81	
LED-128	22.6	13.7	-	23.1	13.3	-	[4]
PRESENT-128	10.8	-	-	10.3	-	-	[4]
Piccolo-128	9.2	-	-	9.2	-	-	[4]

Counter Mode Implementations of SKINNY-64-128.

As mentioned before, we can evaluate the speed of SKINNY-64-128 in the same conditions as the benchmarks provided in [3]. Namely, the goal is to generate the keystream from the counter mode using SKINNY-64-128 as the underlying block cipher. The main difference to the previous scenario is that many blocks of a non-repeating value (counter) are encrypted. This allows to save the costs for data packing, as the values are known in advance and can already be provided in the correct format.

The designers of SIMONachieve a very high performances, by taking advantage of this mode, in their implementation available on GitHub.[7] We would like to note that this CTR-mode implementation does not process the same amount of blocks as given in Table 18 and we expect the performance of SIMONto be closer to these figures for an optimized implementation.

In our case, we devise a very similar implementation that considers 64 blocks in parallel and reaches a maximal speed of 2.63 cpb in the same setting on the latest Intel platform Skylake. We note that the key is pre-expanded prior to encrypting the blocks, and the 64 blocks are stored in 16 registers of 256 bits in a bit-sliced way. In detail, the four first registers contain the four first bits of each first row of the 64 blocks. The same holds for the 12 others registers with the remaining three rows of the states.

Then, for all the 36 rounds of SKINNY-64-128, the application of SubCells, AddConstants, AddRoundTweakey, and MixColumns can be easily done with bit-wise operations on registers. As for ShiftRows, we implement it as a shuffle on bytes within each register. The benchmarks conducted on our four platforms are shown in Table 19.

Table 19: Counter mode implementations of SKINNY-64-128, SKINNY-128-128, SIMON-64-128 and SIMON-128-128. Performances are given in cycles per byte, with pre-expanded subkeys, encrypting 16384 bytes and obtained using Supercop. Details of the machines are given in Table 17.

	Westmere	Ivy Bridge	Haswell		Skylake		Ref.
Instruction Set	sse4	sse4	sse4	avx2	sse4	avx2	
SKINNY-64-128	7.87	5.27	5.14	2.92	4.79	2.63	**New**
SIMON-64-128	7.64	5.85	5.93	3.12	5.26	2.71	[47]
SKINNY-128-128	-	-	7.41	4.05	7.02	3.76	**New**
SIMON-128-128	11.52	8.87	8.70	4.39	7.99	4.00	[47]

[7] Available at https://github.com/lrwinge/simon_speck_supercop/.

5.4 Micro-Controller Implementations

In this section, we describe the main performance figures when implementing SKINNY on micro-controllers. We omit details of the implementation and mainly describe the results of the implementation.

To evaluate the suitability of SKINNY for usage in embedded environments, we implemented SKINNY for the ATmega644 microcontroller (avr5-core). Note that we choose to exclude the C-interface related overhead since this is often depending on the chosen environment and compiler.

The three main indicators for performance are:

1. Speed, measured in cycles per byte. The figures given below correspond to the execution time of the encryption function. It is expected that a pointer to the RAM-residing preprocessed key is passed to the encryption function.
2. RAM size, measured in bytes. This includes all RAM used for the encryption process. Especially global data structures like Sboxes or round constants are included if they are stored in RAM. Also the RAM used to hold the preprocessed key is accounted.
3. ROM size, measured in bytes. This corresponds to the complete footprint of the algorithm needed to initialize global data structures, preprocess the key and to encrypt data. Especially also the memory is accounted for data which is only copied into RAM (known as `.data` segment).

All our implementations allow changing the key at runtime and some of them require the initialisation of global data structures.

A lot of different trade-offs can be made, which is a real strength of SKINNY, since different applications may have very different requirements and total costs would be computed very differently, sometimes justifying sacrifices which would be unacceptable in most cases. In Table 20, we provide several of those trade-offs, each optimizing for another of the three criteria mentioned above.

Table 20: Implementation figures for SKINNY-128-128 on an ATmega644

Cycles per byte	**222**	257	258	288	297	328
RAM	576	287	576	287	**31**	**31**
ROM	676	616	492	**436**	774	594

To compare, for example, with SIMON-128-128 on the same platform, note that, according to [3], SIMON-128-128 can be implemented to optimize speed

with 510 byte of ROM and 544 bytes of RAM running at 337 cycles per byte. Thus, it can be seen that SKINNY-128-128 can be significantly faster (while sacrificing some ROM and RAM size).

6 The Low-Latency Tweakable Block Cipher MANTIS

In this section, we present a tweakable block cipher design which is optimized for low-latency implementations.

The low-latency block cipher PRINCE already provides a very good starting point for a low-latency design. Its round function basically follows the **aes** structure, with the exception of using a MixColumns-like mapping with branch number 4 instead of 5. The main difference between PRINCE and **aes** (and actually all other ciphers) is that the design is symmetric around a linear layer in the middle. This allows to realize what was coined α-reflection: the decryption for a key K corresponds (basically) to encryption with a key $K \oplus \alpha$ where α is a fixed constant. Turning PRINCE into a tweakable block cipher is (conceptually) well understood when using e.g. the TWEAKEY framework [22]. First, define a tweakey-schedule and than simply increase the number of rounds until one can ensure that the cipher is secure against related-tweak attacks.

However, the problem is that the latency of a cipher is directly related to the number of rounds. Thus, it is crucial to find a design, i.e. a round function and a tweak-scheduling, that ensures security already with a minimal number of rounds. Here, components of the recently proposed block ciphers MIDORI [1] turn out to be very beneficial. In MIDORI, again an **aes**-like design, one of the key observations was that changing ShiftRows into a more general permutation allows to significantly improve upon the number of active Sboxes (in the single key model) while keeping a MixColumns-like layer with branch number 4 only. On top, the designers of MIDORI designed a 4-bit Sbox that was optimized with respect to circuit-depth. This directly leads to an improved version of PRINCE itself: replace the PRINCE round function by the MIDORI-round function while keeping the entire design symmetric around the middle to keep the α-reflection property. This simple change would result in a cipher with improved latency and improved security (i.e. number of active Sboxes) compared to PRINCE. It is actually exactly this PRINCE-like MIDORI that we use as a starting point for designing the low-latency block cipher MANTIS. The final step in the design of MANTIS was to find a suitable tweak-scheduling that would ensure a high number of active Sboxes not only in the single-key setting, but also in the setting where the attacker can con-

trol the difference in the tweak. Using, again, the MILP approach, we are able to demonstrate that a slight increase in the number of rounds (from 12 to 14) is already sufficient to ensure the resistance of MANTIS to differential (and linear) attacks in the related-tweak setting. Note that MANTIS is certainly not secure in the related-key model, as there always exist a probability one distinguisher caused by the α-reflection property.

$\texttt{MANTIS}_r$ has a 64-bit block length and works with a 128-bit key and 64-bit tweak. The parameter r specifies the number of rounds of one half of the cipher. The overall design is illustrated in Figure 1.

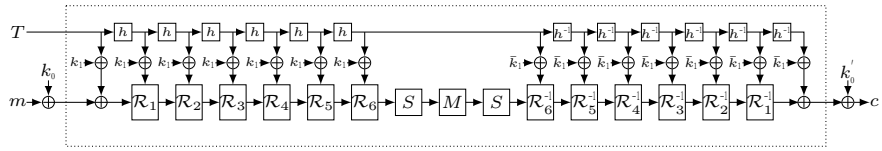


Figure 1: Illustration of $\texttt{MANTIS}_6$.

We acknowledge the contribution of Roberto Avanzi to the design of MANTIS. He first suggested us to combine PRINCE with the TWEAKEY framework, and also to modify the latter by permuting the tweak independently from the key, in order to save on the Galois multiplications of the tweak cells. He then brainstormed with us on early versions of the design.

6.1 Description of the Cipher

$\texttt{MANTIS}_r$ is based on the FX-construction [25] and thus applies whitening keys before and after applying its core components. The 128-bit key is first split into $k = k_0 \parallel k_1$ with 64-bit subkeys k_0, k_1. Then, $(k_0 \parallel k_1)$ is extended to the 192 bit key

$$(k_0 \parallel k_0' \parallel k_1) := (k_0 \parallel (k_0 \ggg 1) \oplus (k_0 \gg 63) \parallel k_1),$$

and k_0, k_0' are used as whitening keys in an FX-construction. The subkey k_1 is used as the round key for all of the $2r$ rounds of $\texttt{MANTIS}_r$. We decided to stick with the FX construction for simplicity., even so other options as described in [12].

Initialization.

The cipher receives a plaintext $m = m_0 \| m_1 \| \cdots \| m_{14} \| m_{15}$, where the m_i are 4-bit cells. The initialization of the cipher's internal state is performed by setting $IS_i = m_i$ for $0 \leqslant i \leqslant 15$.

The cipher also receives a tweak input $T = t_0 \| t_1 \| \cdots \| t_{15}$, where the t_i are 4-bit cells. The initialization of the cipher's tweak state is performed by setting $T_i = t_i$ for $0 \leqslant i \leqslant 15$. Thus,

$$IS = \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix} \quad T = \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ t_4 & t_5 & t_6 & t_7 \\ t_8 & t_9 & t_{10} & t_{11} \\ t_{12} & t_{13} & t_{14} & t_{15} \end{bmatrix}$$

The round function.

One round $\mathcal{R}_i(\cdot, tk)$ of $\texttt{MANTIS}_r$ operates on the cipher internal state depending on the round tweakey tk as

$$\texttt{MixColumns} \circ \texttt{PermuteCells} \circ \texttt{AddTweakey}_{tk} \circ \texttt{AddConstant}_i \circ \texttt{SubCells}.$$

In the following, we describe the components of the round function.

`SubCells`. The involutory `MIDORI` Sbox $\texttt{Sb}_0$ is applied to every cell of the internal state. A description of the Sbox is given in Table 21. Using the `MIDORI` Sbox is beneficial as this Sbox is especially optimized for small area and low circuit depth.

Table 21: 4-bit involutory `MIDORI` Sbox $\texttt{Sb}_0$ used in `MANTIS`.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$\texttt{Sb}_0[x]$	c	a	d	3	e	b	f	7	8	9	1	5	0	2	4	6

`AddConstant`. In the i-th round, the round constant RC_i is XORed to the internal state. The round constants are generated in a similar way as for `PRINCE`, that is we used the first digits of π to generate those constants (actually the very first digits correspond to α defined below). The round constants can be found in Table 22. Note that, in contrast to `PRINCE`, the constants are added row-wise instead of column-wise.

`AddRoundTweakey`. In round $\mathcal{R}_i$, the (full) round tweakey state $h^i(T) \oplus k_1$ is XORed to the cipher internal state. In the i-th inverse round $\mathcal{R}_i^{-1}$, the tweakey state $h^i(T) \oplus \bar{k}_1 := h^i(T) \oplus k_1 \oplus \alpha$ with $\alpha = \texttt{0x243f6a8885a308d3}$ is XORed to the internal state. Note that this α, as the round constants,

Table 22: Round Constants used in MANTIS.

Round i	Round constant RC_i	Round i	Round constant RC_i
1	0x13198a2e03707344	5	0xbe5466cf34e90c6c
2	0xa4093822299f31d0	6	0xc0ac29b7c97c50dd
3	0x082efa98ec4e6c89	7	0x3f84d5b5b5470917
4	0x452821e638d01377	8	0x9216d5d98979fb1b

is chosen as the first digits of π. Thereby, it is $h(T) = t_{h(0)} \| t_{h(1)} \cdot \| t_{h(15)}$, where the tweak permutation h is defined as

$$h = [6, 5, 14, 15, 0, 1, 2, 3, 7, 12, 13, 4, 8, 9, 10, 11].$$

`PermuteCells`. The cells of the internal state are permuted according to the `MIDORI` permutation

$$P = [0, 11, 6, 13, 10, 1, 12, 7, 5, 14, 3, 8, 15, 4, 9, 2].$$

Note that the `MIDORI` permutation ensures a higher number of active Sboxes compared to the choice made in `PRINCE`.

`MixColumns`. Each column of the cipher internal state array is multiplied by the binary matrix used in `MIDORI` and shown below.

$$\mathbf{M} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}.$$

Encryption.

In the following, we define H_r as the application of r rounds $\mathcal{R}_i$ and one additional `SubCells` layer. Similarly, we define H_r^{-1} as the application on one inverse `SubCells` layer plus r inverse rounds. Thus,

$$\begin{aligned} H_r(\cdot, T, k_1) &= \texttt{SubCells} \circ \mathcal{R}_r(\cdot, h^r(T) \oplus k_1) \circ \cdots \circ \mathcal{R}_1(\cdot, h(T) \oplus k_1) \\ H_r^{-1}(\cdot, T, \bar{k}_1) &= \mathcal{R}_1^{-1}(\cdot, h(T) \oplus \bar{k}_1) \circ \cdots \circ \mathcal{R}_r^{-1}(\cdot, h^r(T) \oplus \bar{k}_1) \circ \texttt{SubCells}. \end{aligned}$$

With this notation, it is

$$\begin{aligned} \mathbf{Enc}_{(k_0, k_0', k_1)}(\cdot, T) = \; & \texttt{AddTweakey}_{k_0' \oplus k_1 \oplus \alpha \oplus T} \circ H_r^{-1}(\cdot, T, k_1 \oplus \alpha) \\ & \circ \texttt{MixColumns} \circ H_r(\cdot, T, k_1) \circ \texttt{AddTweakey}_{k_0 \oplus k_1 \oplus T} \end{aligned}$$

Decryption.

It is $\mathbf{Enc}^{-1}_{(k_0,k_0',k_1)}(\cdot,T) = \mathbf{Enc}_{(k_0',k_0,k_1\oplus\alpha)}(\cdot,T)$ because of the α-reflection property.

6.2 Design Rationale

The goal was to design a cipher which is competitive to PRINCE in terms of latency with the advantage of being tweakable. In contrast to SKINNY, we distinguish between tweak and key input. In particular, we allow an attacker to control the tweak but not the key. Thus, similar to PRINCE, we do not claim related-key security. In order to reach this goal, again, several components are borrowed from already existing ciphers. In the following, we present the reasons for our design. Note that, as we aim for an efficient unrolled implementation, one is not restricted to a classical round-iterated design.

α-Reflection Property.

MANTIS_r is designed as a reflection cipher such that encryption under a key k equals decryption under a related key. This significantly reduces the implementation overhead for decryption. Therefore, the parameter r denotes only half the number of rounds, as the second half of the cipher is basically the inverse of the first half. It is advantageous that the diffusion matrix $\mathbf{M}$ is involutory since we need the middle part of the cipher to be an involution. Unlike in the description of PRINCE, we use the same round constant for the inverse $\mathcal{R}_i^{-1}$ of the i-th round and apply the addition of α to the round key k_1.

The Choice of the Diffusion Layer.

To achieve low latency in an unrolled implementation, one is limited in the number rounds to be applied. Therefore, one has to achieve very fast diffusion while guaranteeing a high number of active Sboxes. To reach these requirements, we adopted the linear layer of MIDORI. It provides full diffusion only after three rounds and guarantees a high number of active Sboxes in the single-key setting. We refer to Table 4 for the bounds.

The Choice of the Sbox.

For the Sbox in MANTIS we used the same Sbox as in MIDORI. The MIDORI Sbox has a significantly smaller latency than the PRINCE Sbox. The maximal linear bias is 2^{-2} and the best differential probability is 2^{-2} as well.

The Choice of the Tweakey Permutation h.

Our aim was to choose a tweak permutation h such that five rounds (plus one additional SubCells layer) guarantee at least 16 active Sboxes in the related-tweak setting. This would guarantee at least 32 active Sboxes for $\texttt{MANTIS}_5$ which is enough to bound the differential probability (resp. linear bias) below $2^{-2 \cdot 32}$. Since there are 16! possibilities for h, which is too much for an exhaustive search, we restricted ourself on a subclass of 8! tweak permutations. The restriction is that two complete rows (without changing the position of the cells in those rows) are permuted to different rows. In our case, the first and third row are permuted to the second and fourth row, respectively. The bounds were derived using the MILP tool. We tested several thousand choices for the permutation h and found out that 16 active Sboxes were the best possible to reach over H_5. Out of these optimal choices, we took the permutation that maximized the bound for $\texttt{MANTIS}_5$, and as a second step for $\texttt{MANTIS}_6$. We refer to Table 23 for the actual bounds.

Table 23: Lower bounds on the number of linear (and differential) active Sboxes in the single-key model and of differential active Sboxes in the related-tweak model.

	$\texttt{MANTIS}_2$	$\texttt{MANTIS}_3$	$\texttt{MANTIS}_4$	$\texttt{MANTIS}_5$	$\texttt{MANTIS}_6$	$\texttt{MANTIS}_7$	$\texttt{MANTIS}_8$
Linear	14	32	46	62	70	76	82
Related Tweak	6	12	20	34	44	50	56

Security Claim.

For $\texttt{MANTIS}_7$, we claim that any adversary who in possession of 2^n chosen plain/ciphertext pairs which were obtained under chosen tweaks, but with a fixed unknown key, needs at least 2^{126-n} calls to the encryption function in order to recover the secret key. Thus, our security claims are the same as for PRINCE, except that we also claim related-tweak security. Moreover, already for $\texttt{MANTIS}_5$ we claim security against practical attacks, similar to what has been considered in the PRINCE challenge. More precisely, we claim that no related-tweak attack (better than the generic claim above) is possible against $\texttt{MANTIS}_5$ with less than 2^{30} chosen or 2^{40} known plaintext/ciphertext pairs. Note that because of the α-reflection, there exists a trivial related-key distinguisher with probability one. We especially encourage further cryptanalysis on the aggressive versions.

6.3 Security Analysis

As one round of MANTIS is almost identical to one round in MIDORI, most of the security analysis can simply be copied from the latter. This holds in particular for meet-in-the-middle attacks, integral attacks and slide attacks. We therefore only focus on the attacks where the changes in round constants and by adding the tweak actually result in different arguments.

Invariant Subspaces.

The most successful attack against MIDORI-64 at the moment is an invariant subspace attack with a density of 2^{96} weak keys. The main observation here is that the round constants in MIDORI are too sparse and structured to avoid certain symmetries. More precisely, the round constants in MIDORI-64 only affect a single bit in each of the 16 4-bit cells. Together with a property of the Sbox this finally results in the mentioned attack. For MANTIS, the situation is very different as the round constants (in each half) are basically random values. This in particular ensures that the invariant subspace attack on MIDORI does not translate into an attack on MANTIS.

Differential and Linear Related-Tweak Attacks.

Using the MILP approach, we are able to prove strong bounds against related-tweak linear and differential attacks. In particular, no related tweak linear or differential distinguisher based on a characteristics is possible for $\texttt{MANTIS}_5$, that is already for 12 layers of Sboxes. As $\texttt{MANTIS}_7$ has four more rounds, and additional key-whitening, we believe that is provides a small but sufficient security margin.

6.4 Implementations

In Table 24 and Table 25, we list results of unrolled implementations for MANTIS constrained for the smallest area and the shortest latency respectively. In particular, it can be seen that for $\texttt{MANTIS}_5$, the difference in area compared to PRINCE corresponds quite exactly to the additional costs of the XOR gates needed to add the tweak. However, by constraining the synthesis to a particular latency, $\texttt{MANTIS}_5$ outperforms PRINCE mainly due to its underlying MIDORI Sbox. A complete overview of the latency versus delay for all variants of MANTIS compared to PRINCE is shown in Figure 2.

Table 24: Unrolled implementations of MANTIS constrained for the smallest area (both encryption and decryption).

	Area	Delay	Ref.
	GE	ns	
MANTIS_5	8544	15.95	**New**
MANTIS_6	9861	17.60	**New**
MANTIS_7	11209	20.50	**New**
MANTIS_8	12533	21.34	**New**
PRINCE	8344	16.00	[30]

Table 25: Unrolled implementations of MANTIS constrained for the shortest delay (both encryption and decryption).

	Area	Delay	Ref.
	GE	ns	
MANTIS_5	13424	9.00	**New**
MANTIS_6	18375	10.00	**New**
MANTIS_7	23926	11.00	**New**
MANTIS_8	30252	12.00	**New**
PRINCE	17693	9.00	[30]

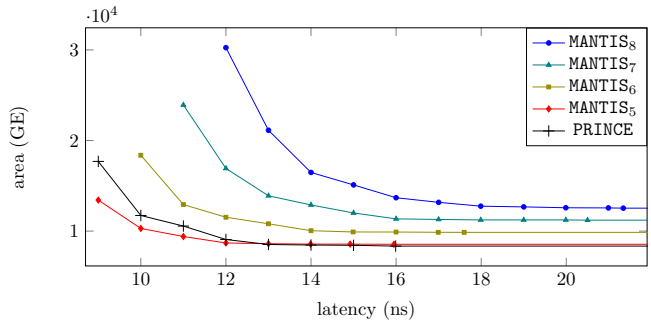


Figure 2: Latency versus area of MANTIS compared to PRINCE.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. This work is partly supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06), the DFG Research Training Group GRK 1817 Ubicrypt and the BMBF Project UNIKOPS (01BY1040). We furthermore like to thank Daniel Otto for providing us with performance figures for SKINNY on micro-controllers.

References

[1] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. "Midori: A Block Cipher for Low Energy." In: *Advances in Cryptology - ASIACRYPT 2015 - Part II*. Vol. 9453. Lecture Notes in Computer Science. Springer, 2015, pp. 411–436. ISBN: 978-3-662-48799-0. DOI: 10.1007/978-3-662-48800-3_17. URL: http://dx.doi.org/10.1007/978-3-662-48800-3_17.

[2] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. *The SIMON and SPECK Families of Lightweight Block Ciphers*. ePrint/2013/404. 2013.

[3] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. *SIMON and SPECK: Block Ciphers for the Internet of Things*. ePrint/2015/585. 2015.

[4] Ryad Benadjila, Jian Guo, Victor Lomné, and Thomas Peyrin. "Implementing Lightweight Block Ciphers on x86 Architectures." In: pp. 324–351. DOI: 10.1007/978-3-662-43414-7_17.

[5] Eli Biham, Alex Biryukov, and Adi Shamir. "Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials." In: *EUROCRYPT 1999*. Vol. 1592. Lecture Notes in Computer Science. Springer, 1999, pp. 12–23.

[6] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Trade-Offs for Threshold Implementations Illustrated on AES." In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 34.7 (2015), pp. 1188–1200. DOI: 10.1109/TCAD.2015.2419623. URL: http://dx.doi.org/10.1109/TCAD.2015.2419623.

[7] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia Tokareva, and Valeriya Vitkup. "Threshold Implementations of Small S-boxes." In: *Cryptography and Communications* 7.1 (2015), pp. 3–33. DOI: 10.1007/s12095-014-0104-7. URL: http://dx.doi.org/10.1007/s12095-014-0104-7.

[8] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. "PRESENT: An Ultra-Lightweight Block Cipher." In: pp. 450–466.

[9] Andrey Bogdanov and Christian Rechberger. "A 3-Subset Meet-in-the-Middle Attack: Cryptanalysis of the Lightweight Block Cipher KTANTAN." In: *SAC 2010*. Vol. 6544. Lecture Notes in Computer Science. Springer, 2010, pp. 229–240.

[10] Julia Borghoff et al. "PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract." In: pp. 208–225. DOI: 10.1007/978-3-642-34961-4_14.

[11] Christina Boura and Anne Canteaut. *Another View of the Division Property*. CRYPTO 2016. LNCS, Springer. to appear, 2016.

[12] Christina Boura, Anne Canteaut, Lars R. Knudsen, and Gregor Leander. *Reflection ciphers*. Designs, Codes and Cryptography. 2015.

[13] Christophe De Cannière, Orr Dunkelman, and Miroslav Knežević. "KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers." In: pp. 272–288.

[14] Anne Canteaut, Sébastien Duval, and Gaëtan Leurent. *Construction of Lightweight S-Boxes using Feistel and MISTY structures (Full Version)*. ePrint/2015/711. 2015.

[15] David Chaum and Jan-Hendrik Evertse. "Crytanalysis of DES with a Reduced Number of Rounds: Sequences of Linear Factors in Block Ciphers." In: *CRYPTO 1985*. Vol. 218. Lecture Notes in Computer Science. Springer, 1985, pp. 192–211.

[16] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. "The Block Cipher Square." In: *FSE '97*. Vol. 1267. Lecture Notes in Computer Science. Springer, 1997, pp. 149–165.

[17] Joan Daemen, Michal Peeters, Gilles Van Assche, and Vincent Rijmen. *Nessie Proposal: the Block Cipher* NOEKEON. Nessie submission. http://gro.noekeon.org/. 2000.

[18] Benoît Gérard, Vincent Grosso, María Naya-Plasencia, and François-Xavier Standaert. "Block Ciphers That Are Easier to Mask: How Far Can We Go?" In: pp. 383–399. DOI: 10.1007/978-3-642-40349-1_22.

[19] Lorenzo Grassi, Sondre Rønjom, and Christian Rechberger. *Subspace Trail Cryptanalysis and its Applications to AES*. ePrint/2016/592. 2016.

[20] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. "The LED Block Cipher." In: pp. 326–341.

[21] Michael Henson and Stephen Taylor. "Memory encryption: A survey of existing techniques." In: *ACM Comput. Surv.* 46.4 (2013), 53:1–53:26. DOI: 10.1145/2566673. URL: http://doi.acm.org/10.1145/2566673.

[22] Jérémy Jean, Ivica Nikolic, and Thomas Peyrin. "Tweaks and Keys for Block Ciphers: The TWEAKEY Framework." In: pp. 274–288. DOI: 10.1007/978-3-662-45608-8_15.

[23] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. *Joltik v1.3*. Submission to the CAESAR competition, http://www1.spms.ntu.edu.sg/~syllab/Joltik. 2015.

[24] Khoongming Khoo, Thomas Peyrin, Axel York Poschmann, and Huihui Yap. "FOAM: Searching for Hardware-Optimal SPN Structures and Components with a Fair Comparison." In: pp. 433–450. DOI: 10.1007/978-3-662-44709-3_24.

[25] Joe Kilian and Phillip Rogaway. "How to Protect DES Against Exhaustive Key Search." In: pp. 252–267.

[26] Lars R. Knudsen and David Wagner. "Integral Cryptanalysis." In: *FSE 2002*. Vol. 2365. Lecture Notes in Computer Science. Springer, 2002, pp. 112–127.

[27] Stefan Kölbl, Gregor Leander, and Tyge Tiessen. "Observations on the SIMON Block Cipher Family." In: pp. 161–185. DOI: 10.1007/978-3-662-47989-6_8.

[28] Thorsten Kranz, Gregor Leander, and Friedrich Wiemer. *Linear Cryptanalysis: On Key Schedules and Tweakable Block Ciphers*. Preprint. 2016.

[29] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. "Pushing the Limits: A Very Compact and a Threshold Implementation of AES." In: *Advances in Cryptology - EUROCRYPT 2011*. Vol. 6632. Lecture Notes in Computer Science. Springer, 2011, pp. 69–88. ISBN: 978-3-642-20464-7. DOI: 10.1007/978-3-642-20465-4_6. URL: http://dx.doi.org/10.1007/978-3-642-20465-4_6.

[30] Amir Moradi and Tobias Schneider. *Side-Channel Analysis Protection and Low-Latency in Action - case study of PRINCE and Midori*. Cryptology ePrint Archive, Report 2016/481. http://eprint.iacr.org/. 2016.

[31] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. "Differential and Linear Cryptanalysis Using Mixed-Integer Linear Programming." In: *Inscrypt 2011*. 2012, pp. 57–76. ISBN: 978-3-642-34704-7. DOI: 10.1007/978-3-642-34704-7_5. URL: http://dx.doi.org/10.1007/978-3-642-34704-7_5.

[32] National Institute of Standards and Technology. *Recommendation for Key Management – NIST SP-800-57 Part 3 Revision 1*. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf.

[33] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. "Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches." In: *J. Cryptology* 24.2 (2011), pp. 292–321. DOI: 10.1007/s00145-010-9085-7. URL: http://dx.doi.org/10.1007/s00145-010-9085-7.

[34] Thomas Peyrin and Yannick Seurin. *Counter-in-Tweak: Authenticated Encryption Modes for Tweakable Block Ciphers*. ePrint/2015/1049. 2015.

[35] Gilles Piret, Thomas Roche, and Claude Carlet. "PICARO - A Block Cipher Allowing Efficient Higher-Order Side-Channel Resistance." In: pp. 311–328.

[36] Axel Poschmann, Amir Moradi, Khoongming Khoo, Chu-Wee Lim, Huaxiong Wang, and San Ling. "Side-Channel Resistant Crypto for Less than 2, 300 GE." In: *J. Cryptology* 24.2 (2011), pp. 322–345. DOI: 10.1007/s00145-010-9086-6. URL: http://dx.doi.org/10.1007/s00145-010-9086-6.

[37] Yu Sasaki. "Meet-in-the-Middle Preimage Attacks on AES Hashing Modes and an Application to Whirlpool." In: *FSE 2011*. Vol. 6733. Lecture Notes in Computer Science. Springer, 2011, pp. 378–396.

[38] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. "Piccolo: An Ultra-Lightweight Blockcipher." In: pp. 342–357.

[39] F-X Standaert, Gilles Piret, Gael Rouvroy, and J-J Quisquater. "FPGA implementations of the ICEBERG block cipher." In: *INTEGRATION, the VLSI journal* 40.1 (2007), pp. 20–27.

[40] Francois-Xavier Standaert, Gael Rouvroy, Jean-Jacques Quisquater, Jean-Didier Legat, et al. "Efficient FPGA Implementations of Block Ciphers KHAZAD and MISTY1." In: *Proceedings of the Third NESSIE Workshop*. 2002.

[41] Siwei Sun, Lei Hu, Ling Song, Yonghong Xie, and Peng Wang. "Automatic Security Evaluation of Block Ciphers with S-bP Structures Against Related-Key Differential Attacks." In: *Inscrypt 2013*. 2014, pp. 39–51. ISBN: 978-3-319-12087-4. DOI: 10.1007/978-3-319-12087-4_3. URL: http://dx.doi.org/10.1007/978-3-319-12087-4_3.

[42] Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi. "TWINE : A Lightweight Block Cipher for Multiple Platforms." In: pp. 339–354. DOI: 10.1007/978-3-642-35999-6_22.

[43] Yosuke Todo. "Structural Evaluation by Generalized Integral Property." In: *EUROCRYPT 2015, Part I*. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 287–314.

[44] Vincent Grosso and Gaëtan Leurent and François-Xavier Standaert and Kerem Varici and Anthony Journault and François Durvaux and Lubos Gaspar and Stéphanie Kerckhof. *SCREAM v3*. Submission to the CAESAR competition. 2015.

[45] Virtual Silicon Inc. 0.18 µm *VIP Standard Cell Library Tape Out Ready, Part Number: UMCL18G212T3, Process: UMC Logic* 0.18 µm *Generic II Technology: 0.18µm*. 2004.

[46] Peter Williams and Rick Boivie. "CPU Support for Secure Executables." In: *TRUST 2011*. Vol. 6740. Lecture Notes in Computer Science. Springer, 2011, pp. 172–187. ISBN: 978-3-642-21598-8. DOI: 10.1007/978-3-642-21599-5_13. URL: http://dx.doi.org/10.1007/978-3-642-21599-5_13.

[47] Louis Wingers. *Software for SUPERCOP Benchmarking of SIMON and SPECK*. https://github.com/lrwinge/simon_speck_supercop. 2015.

A 8-bit Sbox for SKINNY-128

```
/* SKINNY-128 Sbox */
uint8_t S8[256] = {
 0x65,0x4c,0x6a,0x42,0x4b,0x63,0x43,0x6b,0x55,0x75,0x5a,0x7a,0x53,0x73,0x5b
     ,0x7b,
 0x35,0x8c,0x3a,0x81,0x89,0x33,0x80,0x3b,0x95,0x25,0x98,0x2a,0x90,0x23,0x99
     ,0x2b,
 0xe5,0xcc,0xe8,0xc1,0xc9,0xe0,0xc0,0xe9,0xd5,0xf5,0xd8,0xf8,0xd0,0xf0,0xd9
     ,0xf9,
 0xa5,0x1c,0xa8,0x12,0x1b,0xa0,0x13,0xa9,0x05,0xb5,0x0a,0xb8,0x03,0xb0,0x0b
     ,0xb9,
 0x32,0x88,0x3c,0x85,0x8d,0x34,0x84,0x3d,0x91,0x22,0x9c,0x2c,0x94,0x24,0x9d
     ,0x2d,
 0x62,0x4a,0x6c,0x45,0x4d,0x64,0x44,0x6d,0x52,0x72,0x5c,0x7c,0x54,0x74,0x5d
     ,0x7d,
 0xa1,0x1a,0xac,0x15,0x1d,0xa4,0x14,0xad,0x02,0xb1,0x0c,0xbc,0x04,0xb4,0x0d
     ,0xbd,
 0xe1,0xc8,0xec,0xc5,0xcd,0xe4,0xc4,0xed,0xd1,0xf1,0xdc,0xfc,0xd4,0xf4,0xdd
     ,0xfd,
 0x36,0x8e,0x38,0x82,0x8b,0x30,0x83,0x39,0x96,0x26,0x9a,0x28,0x93,0x20,0x9b
     ,0x29,
 0x66,0x4e,0x68,0x41,0x49,0x60,0x40,0x69,0x56,0x76,0x58,0x78,0x50,0x70,0x59
     ,0x79,
 0xa6,0x1e,0xaa,0x11,0x19,0xa3,0x10,0xab,0x06,0xb6,0x08,0xba,0x00,0xb3,0x09
     ,0xbb,
 0xe6,0xce,0xea,0xc2,0xcb,0xe3,0xc3,0xeb,0xd6,0xf6,0xda,0xfa,0xd3,0xf3,0xdb
     ,0xfb,
 0x31,0x8a,0x3e,0x86,0x8f,0x37,0x87,0x3f,0x92,0x21,0x9e,0x2e,0x97,0x27,0x9f
     ,0x2f,
 0x61,0x48,0x6e,0x46,0x4f,0x67,0x47,0x6f,0x51,0x71,0x5e,0x7e,0x57,0x77,0x5f
     ,0x7f,
 0xa2,0x18,0xae,0x16,0x1f,0xa7,0x17,0xaf,0x01,0xb2,0x0e,0xbe,0x07,0xb7,0x0f
     ,0xbf,
 0xe2,0xca,0xee,0xc6,0xcf,0xe7,0xc7,0xef,0xd2,0xf2,0xde,0xfe,0xd7,0xf7,0xdf
     ,0xff
};


/* Inverse SKINNY-128 Sbox */
uint8_t S8_inv[256] = {
  0xac,0xe8,0x68,0x3c,0x6c,0x38,0xa8,0xec,0xaa,0xae,0x3a,0x3e,0x6a,0x6e,0xea
     ,0xee,
  0xa6,0xa3,0x33,0x36,0x66,0x63,0xe3,0xe6,0xe1,0xa4,0x61,0x34,0x31,0x64,0xa1
     ,0xe4,
  0x8d,0xc9,0x49,0x1d,0x4d,0x19,0x89,0xcd,0x8b,0x8f,0x1b,0x1f,0x4b,0x4f,0xcb
     ,0xcf,
  0x85,0xc0,0x40,0x15,0x45,0x10,0x80,0xc5,0x82,0x87,0x12,0x17,0x42,0x47,0xc2
     ,0xc7,
  0x96,0x93,0x03,0x06,0x56,0x53,0xd3,0xd6,0xd1,0x94,0x51,0x04,0x01,0x54,0x91
     ,0xd4,
  0x9c,0xd8,0x58,0x0c,0x5c,0x08,0x98,0xdc,0x9a,0x9e,0x0a,0x0e,0x5a,0x5e,0xda
     ,0xde,
  0x95,0xd0,0x50,0x05,0x55,0x00,0x90,0xd5,0x92,0x97,0x02,0x07,0x52,0x57,0xd2
     ,0xd7,
  0x9d,0xd9,0x59,0x0d,0x5d,0x09,0x99,0xdd,0x9b,0x9f,0x0b,0x0f,0x5b,0x5f,0xdb
     ,0xdf,
  0x16,0x13,0x83,0x86,0x46,0x43,0xc3,0xc6,0x41,0x14,0xc1,0x84,0x11,0x44,0x81
     ,0xc4,
  0x1c,0x48,0xc8,0x8c,0x4c,0x18,0x88,0xcc,0x1a,0x1e,0x8a,0x8e,0x4a,0x4e,0xca
     ,0xce,
```

```
  0x35,0x60,0xe0,0xa5,0x65,0x30,0xa0,0xe5,0x32,0x37,0xa2,0xa7,0x62,0x67,0xe2
     ,0xe7,
  0x3d,0x69,0xe9,0xad,0x6d,0x39,0xa9,0xed,0x3b,0x3f,0xab,0xaf,0x6b,0x6f,0xeb
     ,0xef,
  0x26,0x23,0xb3,0xb6,0x76,0x73,0xf3,0xf6,0x71,0x24,0xf1,0xb4,0x21,0x74,0xb1
     ,0xf4,
  0x2c,0x78,0xf8,0xbc,0x7c,0x28,0xb8,0xfc,0x2a,0x2e,0xba,0xbe,0x7a,0x7e,0xfa
     ,0xfe,
  0x25,0x70,0xf0,0xb5,0x75,0x20,0xb0,0xf5,0x22,0x27,0xb2,0xb7,0x72,0x77,0xf2
     ,0xf7,
  0x2d,0x79,0xf9,0xbd,0x7d,0x29,0xb9,0xfd,0x2b,0x2f,0xbb,0xbf,0x7b,0x7f,0xfb
     ,0xff
};
```

B Test Vectors

B.1 Test Vectors for SKINNY

The keys are given as the concatenation of (up to) three tweakey words: TK1, TK1‖TK2, or TK1‖TK2‖TK3.

```
/* Skinny-64-64 */
Key:        f5269826fc681238
Plaintext:  06034f957724d19d
Ciphertext: bb39dfb2429b8ac7

/* Skinny-64-128 */
Key:        9eb93640d088da63
            76a39d1c8bea71e1
Plaintext:  cf16cfe8fd0f98aa
Ciphertext: 6ceda1f43de92b9e

/* Skinny-64-192 */
Key:        ed00c85b120d6861
            8753e24bfd908f60
            b2dbb41b422dfcd0
Plaintext:  530c61d35e8663c3
Ciphertext: dd2cf1a8f330303c
```

```
/* Skinny-128-128 */
Key:        4
     f55cfb0520cac52fd92c15f37073e93
Plaintext:
     f20adb0eb08b648a3b2eeed1f0adda14
Ciphertext: 22
     ff30d498ea62d7e45b476e33675b74

/* Skinny-128-256 */
Key:        009
     cec81605d4ac1d2ae9e3085d7a1f3
            1
     ac123ebfc00fddcf01046ceeddfcab3
Plaintext:  3
     a0c47767a26a68dd382a695e7022e25
Ciphertext:
     b731d98a4bde147a7ed4a6f16b9b587f

/* Skinny-128-384 */
Key:
     df889548cfc7ea52d296339301797449

     ab588a34a47f1ab2dfe9c8293fbea9a5

     ab1afac2611012cd8cef952618c3ebe8
Plaintext:
     a3994b66ad85a3459f44e92b08f550cb
Ciphertext: 94
     ecf589e2017c601b38c6346a10dcfa
```

B.2 Test Vectors for MANTIS

The keys are given as the concatenation $k_0 \| k_1$.

```
/* MANTIS5 */
Key:         92f09952c625e3e9 d7a060f714c0292b
Tweak:       ba912e6f1055fed2
Plaintext:   3b5c77a4921f9718
Ciphertext:  d6522035c1c0c6c1

/* MANTIS6 */
Key:         92f09952c625e3e9 d7a060f714c0292b
Tweak:       ba912e6f1055fed2
Plaintext:   d6522035c1c0c6c1
Ciphertext:  60e43457311936fd

/* MANTIS7 */
Key:         92f09952c625e3e9 d7a060f714c0292b
Tweak:       ba912e6f1055fed2
Plaintext:   60e43457311936fd
Ciphertext:  308e8a07f168f517

/* MANTIS8 */
Key:         92f09952c625e3e9 d7a060f714c0292b
Tweak:       ba912e6f1055fed2
Plaintext:   308e8a07f168f517
Ciphertext:  971ea01a86b410bb
```

C Comparing Theoretical Performance of Lightweight Ciphers

To simplify the analysis, we omitted the constants in all our computations as it has very little impact on the overall theoretical performance results (only a XOR gate on a few bits is required). We note anyway that SKINNY compares favourably to its competitors on this point since it has very lightweight constants: only 7 constants bits are added per round, and thus only 7 bitwise XORs per round are required.

SKINNY-64-128.

The round function first uses an 4-bit Sbox layer, where each Sbox requires 4 NOR and 4 XOR gates, thus amounting to 1 NOR and 1 XOR per bit of internal state. Then, the ShiftRows layer is basically free, while the MixColumns layer requires 3 XOR gates to update 4 bits, thus amounting to 0.75 XOR per bit of internal state. Only 32 bits of subtweakey is XORed to the internal state every round, which costs 0.5 XOR per bit of internal state. In total, the SKINNY-64-128 round function uses 1 NOR gate and 2.25 XOR gates per

bit of internal state. Regarding the tweakey schedule, the permutation P_T is basically free, but the LFSR update for the TK2 state requires 1 XOR gate per 4-bit updated cell. Since only half of the cells of TK2 are updated every round, this leads to 0.125 XOR gate per bit of internal state. Besides, every round two halves of tweakey words are XORed together to compute the subtweakey value, thus amounting to 0.5 XOR gate per bit of internal state. In total, the SKINNY-64-128 tweakey schedule uses 0.625 XOR gate per bit of internal state.

SKINNY-128-128 **and** SKINNY-128-256.

The reasoning and the computations are exactly the same as for SKINNY-64-128, the only difference being that the LFSR update for the TK2 state in the tweakey schedule now costs 0.5625 XOR gate per bit of internal state for SKINNY-128-256 (since one needs 1 XOR gate per 8-bit updated cell and since only half of the cells of TK2 are updated every round) and does not cost anything for SKINNY-128-128.

Simon-64-128.

The round function uses 32 AND gates and 64 XOR gates per round (the word rotations and the Feistel shift being basically free), which amounts to 0.5 AND and 1 XOR per bit of internal state. Besides, only 32 bits of subkey is XORed to the internal state every round, which costs 0.5 XOR per bit of internal state. In total, the SIMON-64-128 round function uses 0.5 AND gate and 1.5 XOR gate per bit of internal state. Regarding the key schedule, the word rotations are basically free, but one counts 96 XOR gates in total. Thus, the SIMON-64-128 key schedule uses 1.5 XOR gate per bit of internal state.

Simon-128-128 **and** Simon-128-256.

The reasoning and the computations are exactly the same as for SIMON-64-128.

KATAN-64-80.

The round function simply uses 3 AND gates and 6 XOR gates per round, thus amounting to 0.047 NOR and 0.094 XOR per bit of internal state. Regarding the key schedule, each round 3 XOR gates per bit of internal state are required.

PRESENT-128.

The round function first uses an 4-bit Sbox layer, where each Sbox requires 3 AND, 1 OR and 11 XOR gates, which amounts to 1 AND and 2.75 XOR per bit of internal state (we count 3 AND and 1 OR gates to be equivalent to 4 AND gates). The bit permutation layer basically comes for free, but 64 bits of subkey is XORed to the internal state every round, which costs 1 XOR per bit of internal state. In total, the PRESENT-128 round function uses 1 AND gate and 3.75 XOR gates per bit of internal state. Regarding the key schedule, the key state rotation is basically free, but 2 Sboxes are applied to it, which amounts to 0.125 AND and 0.34 XOR per bit of internal state.

PICCOLO-128.

The round function uses an 4-bit Sbox layer, applied twice on half of the state. Since the Sbox requires 4 NOR and 4 XOR gates, this eventually amounts to 1 NOR and 1 XOR per bit of internal state. Then, the word permutation is basically free and the mixing layer applies a diffusion matrix very similar to the **aes** matrix (except it computes in $\mathrm{GF}(2^4)$ instead of $\mathrm{GF}(2^8)$). Computing this matrix requires 72 XOR gates (24 for the matrix coefficients and 48 for the elements sums). Since this matrix is applied twice to the state, this amounts to 2.25 XORs per bit of internal state. Moreover, 32 XOR gates per round are needed for the Feistel construction, which amounts to 0.5 XOR per bit of internal state. Only 32 bits of subkey is XORed to the internal state every round, which costs 0.5 XOR per bit of internal state. In total, the PICCOLO-128 round function uses 1 NOR gate and 4.25 XOR gates per bit of internal state. The key schedule of PICCOLO-128 is basically for free as it only consisting in wiring selecting key material.

Noekeon-128.

The Gamma function of NOEKEONrequires 0.5 NOR, 0.5 AND and 1.75 XOR gates per bit of internal state, and the Theta function requires 3.5 XOR gates per bit of internal state. In total, the round function uses 0.5 NOR, 0.5 AND and 5.25 XOR gates per bit of internal state. The key schedule of the "direct" mode of NOEKEONis basically for free as the key material is used as is. However, the "indirect" mode of NOEKEONconsists in applying the internal cipher to pre-process the key material, thus leading to also a cost of 0.5 NOR, 0.5 AND and 5.25 XOR gates per bit of internal state.

aes-128.

The round function first uses an 8-bit Sbox layer, where each Sbox requires 34 NAND and 80 XOR gates, thus amounting to 4.25 NOR and 10 XOR per bit of internal state (we count a NAND gate to be equivalent to a NOR gate). Then, the ShiftRows layer is basically free, while the MixColumns layer requires to apply a diffusion matrix. Computing this matrix requires 160 XOR gates (64 for the matrix coefficients and 96 for the elements sums). Since this matrix is applied four times to the state, this amounts to 5 XORs per bit of internal state. 128 bits of subkey is XORed to the internal state every round, which costs 1 XOR per bit of internal state. In total, the **aes**-128 round function uses 4.25 NOR gates and 16 XOR gates per bit of internal state. Regarding the key schedule, 4 Sboxes are applied, thus amounting to 1.06 NOR and 2.5 XOR per bit of internal state. Moreover, the linear diffusion in the key schedule requires 1 XOR per bit of internal state. In total, the **aes**-128 key schedule uses 1.06 NOR and 3.5 XOR gates per bit of internal state.

aes-256.

The reasoning and the computations are exactly the same as for **aes**-128, except that the key schedule is exactly twice more costly.

Estimated Theoretical Throughput Quality Grade and Area Quality Grade.

From these numbers of gates per round per bit, we can simply compute the total number of gates per bit of internal state (with or without the key schedule). This will give us some indication on the theoretical ranking of the various functions studied regarding their throughput. Moreover, by using the estimations from Section 3.1, we can evaluate the theoretical ranking of the various functions studied regarding their ASIC area in a round-based implementation.

D Computing Active S-Boxes using MILP and Diffusion Test

To evaluate the resistance of our proposals in terms of differential crpytanalysis, we rely on mixed-integer linear programming (MILP) to model the cipher operations. The goal of the MILP problem consists in maximizing the objective function, which counts the number of active Sboxes in a given number of rounds of the primitive.

To describe the model for SKINNY, we introduce the following binary decision variables:

- $\{\bar{x}_{i,j,k} \mid i,j \in \mathbb{Z}_4, k \in \mathbb{Z}_{r+1}\}$ indicate the activity pattern of the S-boxes. In particular, it is $\bar{x}_{i,j,k} = 1$ if and only if the s-box in row i and column j is active in round k.
- $\{\bar{y}_{i,j,k} \mid i,j \in \mathbb{Z}_4, k \in \mathbb{Z}_r\}$ indicate the activity pattern after application of the AddRoundTweakey layer.
- $\{\bar{\kappa}_{i,j} \mid i,j \in \mathbb{Z}_4\}$ indicate the activity pattern of the initial tweakey state.
- We need two sets of auxillary variables, $\{d^{\oplus}_{i,j,k} \mid i \in \mathbb{Z}_2, j \in \mathbb{Z}_4, k \in \mathbb{Z}_r\}$ for the AddRoundTweakey layer and $\{d_{j,k}, d'_{j,k}, d''_{j,k} \mid j \in \mathbb{Z}_4, k \in \mathbb{Z}_r\}$ for the MixColumns layer.

As the AddRoundTweakey and MixColumns layers only consist of wordwise XOR operations, the main building blocks of the model are the particular linear constraints on the XOR operations. For shorter notations, we define the following sets.

Constraints for XOR. We define by $\mathcal{C}_{\oplus}[i_1, i_2, o, d]$ the set of linear constraints

$$\{i_1 \leqslant d\} \cup \{i_2 \leqslant d\} \cup \{o \leqslant d\} \cup \{i_1 + i_2 + o \geqslant 2d\}.$$

Constraints for Mixing. Similarly, by $\mathcal{C}_{\mathbf{M}}[i_1, i_2, i_3, i_4, o_1, o_2, o_3, o_4, d_1, d_2, d_3]$ we define the set of linear constraints

$$\mathcal{C}_{\oplus}[i_1, i_3, o_4, d_1] \cup \mathcal{C}_{\oplus}[o_4, i_4, o_1, d_2] \cup \mathcal{C}_{\oplus}[i_2, i_3, o_3, d_3] \cup \{o_2 = i_1\}.$$

For **SK**, we have to optimize the following MILP model:

Minimize

$$\sum_{i,j \in \mathbb{Z}_4} \sum_{k \in \mathbb{Z}_r} \bar{x}_{i,j,k}$$

Subject to:

1. Excluding the trivial solution

$$\{\textstyle\sum_{i,j \in \mathbb{Z}_4} \bar{x}_{i,j,0} \geqslant 1\}$$

2. Application of the linear layer

$$\bigcup_{k \in \mathbb{Z}_r} \cup_{j \in \mathbb{Z}_4} \; \mathcal{C}_{\mathbf{M}}[\bar{\mathbf{x}}_{P^{-1}(\cdot,j),k}, \bar{\mathbf{x}}_{(\cdot,j,k+1}, d_{j,k}, d'_{j,k}, d''_{j,k}]$$

Thereby,

$$\bar{\mathbf{x}}_{(\cdot),j,k+1} := \left(\bar{x}_{0,j,k+1}, \bar{x}_{1,j,k+1}, \bar{x}_{2,j,k+1}, \bar{x}_{3,j,k+1}\right)$$

$$\bar{\mathbf{x}}_{P^{-1}(\cdot,j),k} := \left(\bar{x}_{P^{-1}(0,j),k}, \bar{x}_{P^{-1}(1,j),k}, \bar{x}_{P^{-1}(2,j),k}, \bar{x}_{P^{-1}(3,j),k}\right)$$

For **TK1**, we have to optimize the following MILP model:

Minimize

$$\sum_{i,j\in\mathbb{Z}_4}\sum_{k\in\mathbb{Z}_r}\bar{x}_{i,j,k}$$

Subject to

1. Excluding the trivial solution

$$\{\textstyle\sum_{i,j\in\mathbb{Z}_4}\bar{x}_{i,j,0}+\bar{\kappa}_{i,j}\geqslant 1\}$$

2. Application of the TWEAKEY addition to half of the state

$$\bigcup_{k\in\mathbb{Z}_r}\cup_{i\in\{0,1\}}\cup_{j\in\mathbb{Z}_4}\ \mathcal{C}_{\oplus}[\bar{x}_{i,j,k},\bar{\kappa}_{P_T^k(i,j)},\bar{y}_{i,j,k},d^{\oplus}_{i,j,k}]\qquad\cup$$
$$\cup_{i\in\{2,3\}}\cup_{j\in\mathbb{Z}_4}\ \{\bar{y}_{i,j,k}=\bar{x}_{i,j,k}\}$$

3. Application of the linear layer

$$\bigcup_{k\in\mathbb{Z}_r}\cup_{j\in\mathbb{Z}_4}\ \mathcal{C}_{M}[\bar{x}_{P^{-1}(\cdot,j),k},\bar{x}_{(\cdot),j,k+1},d_{j,k},d'_{j,k},d''_{j,k}]$$

Thereby,

$$\bar{x}_{(\cdot),j,k+1}:=\left(\bar{x}_{0,j,k+1},\bar{x}_{1,j,k+1},\bar{x}_{2,j,k+1},\bar{x}_{3,j,k+1}\right)$$
$$\bar{x}_{P^{-1}(\cdot,j),k}:=\left(\bar{x}_{P^{-1}(0,j),k},\bar{x}_{P^{-1}(1,j),k},\bar{x}_{P^{-1}(2,j),k},\bar{x}_{P^{-1}(3,j),k}\right)$$

On The Tightness of the MILP Bounds.

The solution of these models determines a lower bound on the number of differential active Sboxes for any (non-trivial) r-round characteristic in the **SK**, resp. **TK1** scenario. If we consider the word-wise application of the Sbox as a black box, all of the computed bounds for **SK** are tight in the sense that one can construct a valid differential characteristic for a specific choice of Sboxes. In other words, the bound is tight if the Sbox can be chosen independently for every cell and every round. This is less clear in the related-tweakey scenario. So, in this case, we only claim lower bounds and the actual number of active Sboxes might be even better.

Developing New MILP Modeling for TK2 and TK3.

For **TK2** and **TK3**, the model for round function is the same as **TK1**. The main difference from **TK1** is that the cancellation of difference occurs in active cells in the tweakey words, and this must be modeled properly. We stress that this is completely non-trivial, and in fact there has not been proposed any MILP modeling to deal with **TK2** and **TK3**. In this section we, for the first time

in the symmetric-key cryptography community, develop the MILP model to deal with **TK2**, **TK3**, and more generally **TK**x for an integer x.

The difficulty lies in the property when we simulate the result of XORing each tweakey state. For example in **TK2**, the i-th cell of the round tweakey RK[i] is computed by TK1[i] $\oplus$ TK2[i]. With the standard method, we model this XOR with

$$\{a \leqslant d\} \cup \{b \leqslant d\} \cup \{c \leqslant d\} \cup \{a + b + c \geqslant 2d\},$$

where a, b, c are binary variables to denote active/inactive of RK[i], TK1[i], TK2[i] and d is a dummy variable. However, if both of TK1[i] and TK2[i] are active, i.e. their values are 1, the model allows to cancel the difference, and this continues for the entire rounds. Namely, as long as the same cell position in TK1 and TK2 are active, difference will never be propagated into the data processing part.

A bad argument in the above discussion is that it ignores the fact that once TK1[i] = TK2[i] holds, they never cancel each other for a certain number of rounds because the value of TK1[i] is not updated while the value of TK2[i] is update by LFSR. This fact shows that by following the cell-wise method in previous work, MILP cannot return any meaningful lowerbounds. However, converting the cell-wise model into bit-wise model is quite costly, and the model quickly reaches infeasible runtime, especially for 128-bit block version of SKINNY.

Here, our approach is modeling the extracted property of TWEAKEY update instead of modeling the exact specification. First, we focus on a cell in TK1[i] and a cell in TK2[i] which are located in the same cell position. Suppose that X and Y are differences of those two cells. Those cells are XORed to generate a round-key cell in every two rounds. Thus, the equation for a round-key cell in each round becomes as follows.

$$\begin{array}{ll}
\text{Round 1: } X \oplus Y, & \text{Round 2: not generated,} \\
\text{Round 3: } X \oplus \mathrm{LFSR}(Y), & \text{Round 4: not generated,} \\
\text{Round 5: } X \oplus \mathrm{LFSR}^2(Y), & \text{Round 6: not generated,} \\
\text{Round 7: } X \oplus \mathrm{LFSR}^3(Y), & \text{Round 8: not generated,} \\
\cdots & \cdots \\
\text{Round 29: } X \oplus \mathrm{LFSR}^{14}(Y), & \text{Round 30: not generated.}
\end{array}$$

The LFSR has cycle length 15, namely, $Y = \mathrm{LFSR}^{15}(Y)$ and $\mathrm{LFSR}^i(Y) \neq \mathrm{LFSR}^j(Y)$ for all $0 \leqslant i, j \leqslant 14$ such that $i \neq j$. As a result, it is ensured that cancellation between two tweakey states occurs at most once up to round 30 for each cell.[8]

[8] Note that the LFSR is clocked every two rounds.

To model this property, we first define 16 binary variables $\mathrm{LANE}_0, \ldots, \mathrm{LANE}_{15}$, which indicates whether the i-th cell in the initial state is active in at least one of the tweakey states TK1 and TK2. Note that LANE_i is 0 only if both of TK1[i] and TK2[i] are 0. We then also define 16 binary variables representing active/inactive for each round key (results of XORing TK1 and TK2), i.e.

$$\begin{aligned} \mathrm{tk}_0, \mathrm{tk}_1, \ldots, \mathrm{tk}_{15} &\quad \text{for Round 1,} \\ \mathrm{tk}_{16}, \mathrm{tk}_{17}, \ldots, \mathrm{tk}_{31} &\quad \text{for Round 2,} \\ \ldots &\quad \ldots \\ \mathrm{tk}_{16r-16}, \mathrm{tk}_{16r-15}, \ldots, \mathrm{tk}_{16r-1} &\quad \text{for Round } r. \end{aligned}$$

Cell positions will change after the tweakey permutation is applied in each round. For example, the position of LANE_0 corresponds to tk_0 for Round 1, tk_{24} for Round 2, tk_{34} for Round 3, tk_{58} for Round 4, and so on. If $\mathrm{LANE}_0 = 0$, all of these $\mathrm{tk}_0, \mathrm{tk}_{24}, \mathrm{tk}_{34}, \ldots, \mathrm{tk}_{r'}$ must be 0, where $16r-16 \leqslant r' \leqslant 16r-1$. If $\mathrm{LANE}_0 = 1$, at least $r-1$ of $\mathrm{tk}_0, \mathrm{tk}_{24}, \mathrm{tk}_{34}, \ldots \mathrm{tk}_{r'}$ are 1 because number of cancellations is upperbounded by 1 during the first 30 rounds. In the end, we obtain the following constraints for LANE_0;

$$\begin{aligned} \mathrm{tk}_0 - \mathrm{LANE}_0 &\geqslant 0, \\ \mathrm{tk}_{24} - \mathrm{LANE}_0 &\geqslant 0, \\ \mathrm{tk}_{34} - \mathrm{LANE}_0 &\geqslant 0, \\ &\ldots \\ \mathrm{tk}_{r'} - \mathrm{LANE}_0 &\geqslant 0, \\ \mathrm{tk}_0 + \mathrm{tk}_{24} + \mathrm{tk}_{34} + \cdots + \mathrm{tk}_{r'} - r \cdot \mathrm{LNAE}_0 &\leqslant -1. \end{aligned}$$

By generating the constraints similarly for all the LANE_i, one can properly handle the cancellation of tweakey state cells.

For **TK3**, the difference in comaparison to **TK2** is the number of maximum cancellations within 30 rounds, where a cancellation can occur at most twice for each LANE_i. Thus, **TK3** can be modeled by modifying the last inequality by:

$$\mathrm{tk}_0 + \mathrm{tk}_{24} + \mathrm{tk}_{34} + \cdots + \mathrm{tk}_{r'} - r \cdot \mathrm{LNAE}_0 \leqslant -2.$$

Moreover, the general case **TK**x can be modeled by replacing the right hand side of the last inequality by $x-1$.

Diffusion Test.

The cipher *achieves full diffusion after* r *rounds* if every bit of the internal state after the application of r rounds depends on every input bit. For a word-

oriented SPN like SKINNY, the diffusion properties depend both on the linear layer and on the Sbox. Let s denote the word length of the Sbox. To compute these properties, we define the *diffusion matrix* $\mathbf{D}_s$ as described in the following.

$\mathbf{D}_s$ is a 16×16 block matrix which consits of blocks of size $s \times s$ with binary entries. Since SKINNY applies a word-wise binary diffusion matrix and a cell permutation as the linear layer, one can express the linear layer as a binary 16×16 matrix $\mathbf{L}$. In particular,

$$\mathbf{L} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}.$$

Furthermore, for an Sbox $\mathcal{S}$, we define the *dependency matrix* $\mathbf{Dep}(\mathcal{S})$ as

$$\mathbf{Dep}(\mathcal{S})_{i,j} = \begin{cases} 1 \text{ if } \exists x : \mathcal{S}_i(x) \neq \mathcal{S}_i(x + e_j) \\ 0 \text{ else} \end{cases}.$$

Thereby, $\mathcal{S}_i$ denotes the i-th coordinate function and e_j the j-th unit vector. In particular, for the SKINNY Sboxes we have

$$\mathbf{Dep}(\mathcal{S}_4) = \begin{pmatrix}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 1
\end{pmatrix},$$

$$\mathbf{Dep}(\mathcal{S}_8) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Now, we can define the diffusion matrix $\mathbf{D}_s$ for $s \in \{4, 8\}$ as a 16×16 block matrix such that

$$\mathbf{D}_{s_{i,j}} = \begin{cases} \mathbf{Dep}(\mathcal{S}_s) & \text{if } \mathbf{L}_{i,j} = 1 \\ \mathbf{o}_s & \text{if } \mathbf{L}_{i,j} = 0 \end{cases},$$

where $\mathbf{o}_s$ denotes the all-zero matrix of dimension $s \times s$.

Now, the cipher achieves full diffusion after r rounds, if $\mathbf{D}_s^r$ contains no zero entry when $\mathbf{D}_s$ is interpreted as a $16s \times 16s$ matrix over the integers. In this case, every bit of the internal state after r rounds will depend on every input bit.

For `SKINNY-64` and `SKINNY-128`, we made sure that full diffusion is achieved after 6 rounds, both in forward direction and for the inverse. Note that the diffusion matrix of the inverse has to be computed seperately.